



Python

Módulo Básico



Este material foi produzido como parte do projeto de colaboração entre a empresa Huawei Brasil e o Centro Estadual de Educação Tecnológica Paula Souza, representado pela Fatec de Sorocaba e Fatec de Jundiaí

- 2024 -

Recursos deste Curso

Além deste E-Book, este curso conta um amplo conjunto de materiais, nos formatos:

- Vídeo: que demonstram a implementação de programas;
- Código Fonte: que contém as soluções de os exercícios resolvidos e propostos neste E-Book.

Ícone de Vídeo

Neste E-Book haverá uma sinalização quando um vídeo estiver disponível. Esta sinalização é feita com este ícone ao lado. Então você já sabe: encontrou este ícone, há um vídeo disponível.

Padrão gráfico dos Exemplos e Exercícios Resolvidos

Os Exemplos e Exercícios Resolvidos deste E-Book tem o padrão gráfico mostrado a seguir.

Exercício Resolvido nº



Código do programa

Execução

As linhas mais escuras representam o código do programa e as linhas mais claras na sequência representam o resultado que será visível na tela quando o programa for executado.

Dois ambientes de desenvolvimento (IDE) serão usados no desenvolvimento de exemplos e exercícios: Idle e PyCharm.

Todos os Exercícios Resolvidos serão feitos com uso do PyCharm.

Os Exemplos serão feitos com um ou outro. Quando forem feitos com Idle haverá a indicação disso com a frase:

(exemplo interativo feito com IDE Idle)

Sumário

Capítulo 1 O Python	1
1.1 A Linguagem Python	1
1.1.1 Principais características de Python	2
1.1.2 Versões da Linguagem Python.....	3
1.2 Instalação de Python 3	4
1.3 Ambientes de Desenvolvimento (IDE) para Python	5
1.3.1 IDEs off-line.....	5
1.3.2 IDEs On-Line.....	5
1.4 Documentação e Suporte ao Python 3.....	6
1.5 Requisitos Mínimos para rodar os Exemplos deste material	7
1.6 Elementos Fundamentais de Python.....	8
1.7 Comentários	9
1.8 O Zen do Python	10
Capítulo 2 Classes e Objetos.....	11
2.1 Conceitos de Classes e Objetos	11
2.1.1 Armazenamento de dados em programas - Variáveis	11
2.1.2 Tipos de dados	12
2.1.3 Modelo de Dados de Python	12
2.1.4 Objetos simples	16
2.1.5 Objetos compostos.....	16
2.2 Comando de Atribuição.....	18
2.3 Modelo de Dados de Python - Aprofundamento	21
2.3.1 Objetos imutáveis.....	22
2.3.2 Objetos mutáveis.....	22
2.4 Atribuição múltipla	23
2.5 Expressões Aritméticas.....	23
2.5.1 Conceitos iniciais	23
2.5.2 Expressões aritméticas com múltiplos operandos	25
2.5.3 Comando de atribuição incremental	25
2.6 Funções matemáticas	26
Capítulo 3 Comandos de Saída e Entrada.....	28
3.1 A função <code>print()</code>	28
3.1.2 Formatação usando o método <code>.format()</code>	30

3.1.3 Formatação usando f-string	31
3.1.4 O que é o \n ?	32
3.2 A função input ()	33
3.2.1 Funções de conversão de objetos de classes simples	34
3.2.2 Uso conjunto do input com funções de conversão	34
Capítulo 4 Comando Condicional	37
4.1 Conceito geral de um Comando Condicional	37
4.1.1 Conceito geral sobre o Comando Condicional	37
4.1.2 Entendendo os detalhes do exemplo 4.2	39
4.1.3 Indentação	39
4.2 Condições Simples	40
4.3 Negações e Condições Compostas	40
4.4 Condições Compostas Mistas	42
4.5 Comando Condicional – Forma Completa	42
4.6 Comandos condicionais aninhados	43
4.7 Exercícios resolvidos com if-else	45
Capítulo 5 Comandos de Repetição	51
5.1 O comando while.....	51
5.1.1 Como implementar um laço de repetição em Python	51
5.1.2 Fluxo de execução de laços de repetição while	52
5.2 Exercícios resolvidos com while	53
5.3 Mais detalhes sobre laços em Python	56
5.3.1 Comando continue	56
5.3.2 Comando break	57
5.3.3 Cláusula else do comando while	57
Capítulo 6 Tratamento de Exceções	60
6.1 Conceito.....	60
6.2 Tratamento de exceções em Python – Forma essencial	62
6.2.1 Exceções nomeadas.....	62
6.3 Tratamento de exceções em Python – Forma completa.....	64
Capítulo 7 Objetos compostos de Python – Objetos sequenciais	65
7.1 Listas – classe list	65
7.2 Operações com listas.....	67
7.2.1 Criação de uma lista do zero	67
7.2.2 Formas de indexação	67
7.2.3 Eliminação de elementos usando a função del	68
7.2.4 Fatiamento de listas	68
7.2.5 Cópia de uma lista usando fatiamento	69

7.2.6 Métodos da classe <code>list</code>	71
7.2.7 operador <code>in</code>	72
7.3 A Classe <code>range</code>	73
7.4 O Comando <code>for</code>	75
7.4.1 Conceito.....	75
7.4.2 Uso combinado de <code>range</code> e <code>for</code>	76
7.5 Operadores de Concatenação e Multiplicação aplicado a classes sequenciais	77
7.5.1 Concatenação com <code>+</code>	77
7.5.2 Multiplicação com <code>*</code>	77
7.6 Exercícios usando listas	78
7.7 Tuplas – classe <code>tuple</code>	83
7.8 Strings – classe <code>str</code>	85
7.9 Características comuns às classes de sequências – Listas, Tuplas e Strings.....	88

Capítulo 1

O PYTHON

1.1 A LINGUAGEM PYTHON

Python foi concebida no início dos anos 1990, tendo sido publicada em 1991 como projeto pessoal de Guido van Rossum. Trata-se de uma linguagem de programação de computadores que atualmente vem sendo mantida desenvolvida e aprimorada pela Python Software Foundation, uma instituição sem fins lucrativos criada em 2001 e detentora legal dos direitos de propriedade sobre a linguagem.

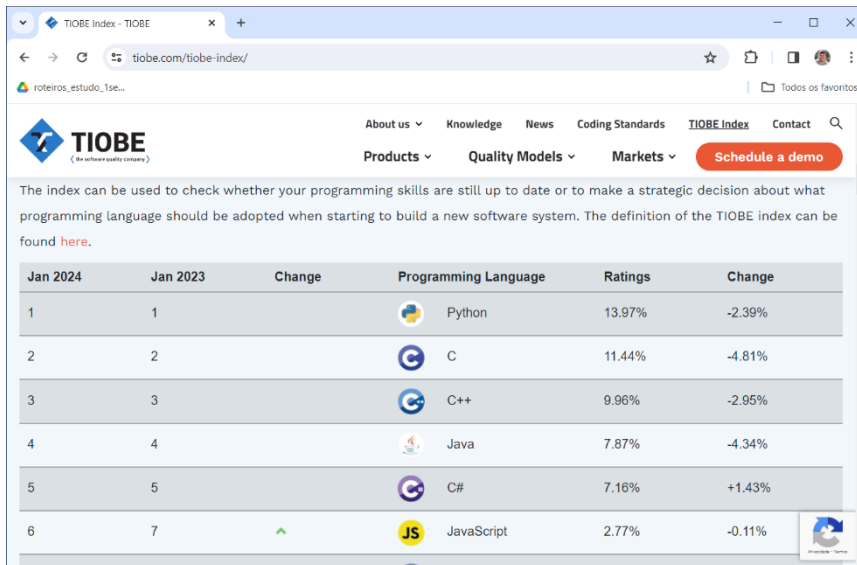
A comunidade mundial de colaboradores e usuários de Python é muito grande e dinâmica. Por um lado, a linguagem é simples e intuitiva, por outro, é poderosa e robusta. Aliar características assim não é nada fácil e em Python isso tem sido alcançado com grande êxito.

O fato de ser simples e intuitiva permite que ela atenda bastante bem ao propósito de ser uma linguagem utilizada por estudantes de programação que precisam de uma ferramenta para realizar seus estudos e implementar seus primeiros algoritmos.

Por ser poderosa e robusta ela pode ser adotada por profissionais de programação que necessitem de uma ferramenta que traga produtividade e confiabilidade aos projetos de software, em especial quando se leva em conta a vasta gama de bibliotecas disponíveis e aplicáveis às várias necessidades frequentemente encontradas no desenvolvimento de software.

Sua importância nos tempos atuais pode ser atestada pela presença no primeiro lugar do índice Tiobe. Este índice é uma lista das linguagens de programação produzida a partir de estatísticas de sites de buscas na internet usando o nome da linguagem como palavra-chave. Como esclarecido no próprio site Tiobe, o índice não existe para indicar qual é a melhor linguagem. Não, não é disso que se trata. O índice Tiobe foi formulado com o propósito de refletir a grau de interesse pelas linguagens de programação ao redor do mundo. Para isso seus principais parâmetros são o número de desenvolvedores que usam as linguagens listadas, a quantidade cursos disponíveis em cada uma, as vagas de emprego ofertados, entre outros.

Figura 1.1 - Índice TIOBE com a linguagem Python em primeiro lugar



The screenshot shows the TIOBE Index website. At the top, there's a navigation bar with links like 'About us', 'Knowledge', 'News', 'Coding Standards', 'TIOBE Index', and 'Contact'. Below this, a table lists the top programming languages. Python is at the top, followed by C, C++, Java, C#, and JavaScript.

	Jan 2024	Jan 2023	Change	Programming Language	Ratings	Change
1	1			Python	13.97%	-2.39%
2	2			C	11.44%	-4.81%
3	3			C++	9.96%	-2.95%
4	4			Java	7.87%	-4.34%
5	5			C#	7.16%	+1.43%
6	7			JavaScript	2.77%	-0.11%

fonte: www.tiobe.com/tiobe-index – acesso em 05 jan 2024

1.1.1 PRINCIPAIS CARACTERÍSTICAS DE PYTHON

Portabilidade

O interpretador Python, bem como suas bibliotecas padrão, estão disponíveis para praticamente todas as plataformas de hardware/sistema operacional, incluindo Unix, Linux, Windows (todas as versões), Mac OS, BeOS, VMS, entre outros. Isto significa que um programa escrito em Python e que use apenas as bibliotecas padrão será executado da mesma maneira em qualquer uma dessas plataformas. Se o software utilizar bibliotecas não consideradas como padrão será necessário verificar a disponibilidade dessas bibliotecas para as várias plataformas.

Código Livre (Opensource)

O código fonte de Python é escrito em linguagem C e disponibilizado como software livre através de uma licença compatível com a GPL. Isso significa que pode ser usado sem a obrigação de qualquer tipo de pagamento de licenças ou *royalties*. Ele pode ser usado por um programador para desenvolver e distribuir um software, assim como seu código fonte pode ser baixado, adaptado e livremente utilizado, uma vez respeitados o que é estabelecido em sua declaração de licença.

Para saber mais sobre a licença de Python acesse este link
<https://docs.python.org/3/license.html>

Simplicidade com robustez

A sintaxe dos comandos de Python é elegante, robusta, simples, legível e poderosa. O núcleo da linguagem conta com um pequeno número de elementos essenciais, coerentes e suficientes. Tais recursos a equiparam a linguagens como C, C++ e Java, permitindo o desenvolvimento de grandes projetos que podem ser orientados a objetos, constituídos por diversos módulos, acessem bancos de dados, trafeguem dados através de redes, trabalhem com recursos multimídia, entre outros. Python também possui mecanismos que permitem a integração com softwares escritos em outras linguagens.

Grande Aplicabilidade

Python pode ser utilizada em um grande número áreas do desenvolvimento de software, das quais se destacam: ferramentas para administração e interface com sistemas operacionais; aplicações que trabalhem com grandes volumes de dados (big-data) armazenados em bancos de dados Sql e NoSql; aplicações gráficas e multimídia; análise de dados, tanto exploratória quanto inferencial; aplicações de machine-learning e inteligência artificial em geral; programação para internet; desenvolvimento de softwares específicos para áreas como estatística, engenharia, biologia, economia e aplicações científicas em geral.

1.1.2 VERSÕES DA LINGUAGEM PYTHON

Existem duas versões de Python que coexistem, conhecidas de maneira generalizada como Python 2 e Python 3.

Embora ambas sejam Python é como se fossem linguagens diferentes. E mesmo que a maioria dos elementos estejam disponíveis nas duas versões, há diferenças significativas a ponto de a versão 3 representar importante quebra de compatibilidade em relação à versão 2. Sobre isso Guido van Rossum declarou em uma postagem de 2007 (ROSSUM, 2007):

Por um bom tempo não havia muito mais do que uma lista de arrependimentos e defeitos estruturais que eram impossíveis de corrigir sem quebrar a compatibilidade retroativa. A ideia era que Python 3 seria o primeiro release do Python a desistir desta compatibilidade em favor de tornar-se uma linguagem melhor e evoluir.

Uma das características mais marcantes da comunidade de desenvolvedores Python é seu conservadorismo em relação a mudanças que possam causar incompatibilidade retroativa. O lançamento de Python 3.0 em 2008 foi um evento singular cercado de grande cuidado, atenção e muitas horas de testes.

Como a quantidade de aplicações e, principalmente, bibliotecas desenvolvidas em Python 2 era muito grande, aliado ao fato de que projetos de migração de software são custosos e complexos, resultou que a adoção da versão 3 foi lenta e enfrentou resistências no início. A equipe de desenvolvimento da linguagem se viu diante de um cenário em que deveria manter dois projetos paralelos, oferecendo suporte e aprimoramento das duas versões.

Em um horizonte longo de tempo isso poderia se tornar um problema gigante e até insolúvel de modo que depois de algum tempo após o lançamento da versão 3 foi publicada no website oficial de Python - www.python.org - a seguinte afirmação: "**Python 2.x é legado, Python 3.x é o presente e o futuro da linguagem**", indicando que a evolução da versão 2 deixaria de existir de fato. Campanhas incentivando a migração foram iniciadas e isso fez com que, aos poucos, Python 3 passasse a ser usada com maior intensidade.

Com o tempo, as vantagens da versão 3 se impuseram. Foram necessários mais de dez anos, mas Python 3 se tornou majoritariamente usada e o suporte a Python 2 encerrou-se oficialmente em 20 de abril de 2020 quando ocorreu o lançamento da derradeira versão 2.7.18.

Neste material serão apresentados exclusivamente os elementos de Python 3.

1.2 INSTALAÇÃO DE PYTHON 3

Linux

Se você usa o sistema operacional Linux não é necessário fazer nada pois a maioria das distribuições contém o Python pré-instalado. Você poderá descobrir qual é a versão que está instalada usando o comando

```
$ which python
```

que retornará a pasta e a versão de instalação. Caso deseje instalar outra versão é só baixá-la e fazer a instalação.

Mac

Em alguns casos, usuários de computadores Mac da Apple podem também ter uma versão padrão do Python pré-instalado. Para verificar isso, basta abrir o Terminal e digitar o comando “python –version”. Se o Python estiver instalado, você verá a versão atual do Python sendo exibida.

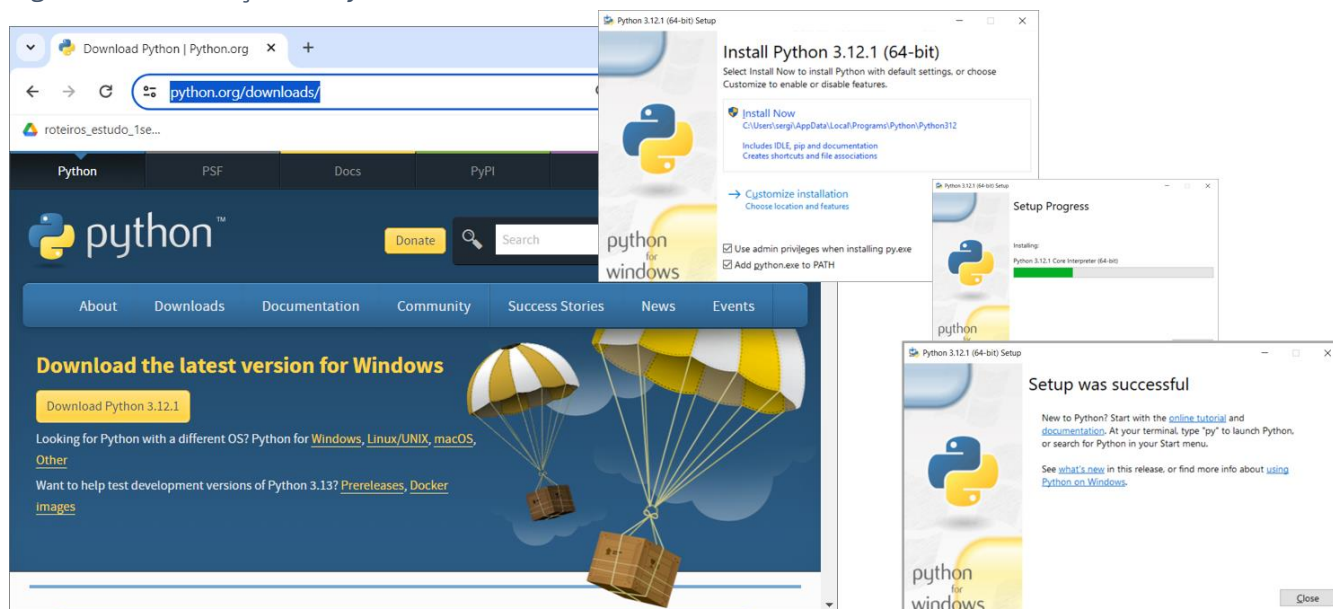
Windows

Em todas as versões do Windows é necessário fazer a instalação de Python 3. E esse é um processo fácil, rápido, seguro, que pode ser feito em poucos minutos. A única restrição é que você terá que possuir um acesso ao Windows com privilégio de administrador.

Acesse o endereço de download e baixe a versão mais recente salvando o instalador em alguma pasta. Inicie a execução do instalador: certifique-se de ligar a opção "Add python.exe to PATH" e depois clique em "Install Now". Aguarde a instalação terminar. Quando isso acontecer já será possível usar a linguagem.

Link para a página de download do Python 3
<https://www.python.org/downloads/>

Figura 1.2 - Instalação de Python 3



fonte: o Autor

Junto com este material está disponível um vídeo demonstrando o download e a instalação de Python 3

Neste texto, sempre que você encontrar este ícone  significa que há um vídeo disponível sobre o assunto.

1.3 AMBIENTES DE DESENVOLVIMENTO (IDE) PARA PYTHON

1.3.1 IDEs OFF-LINE

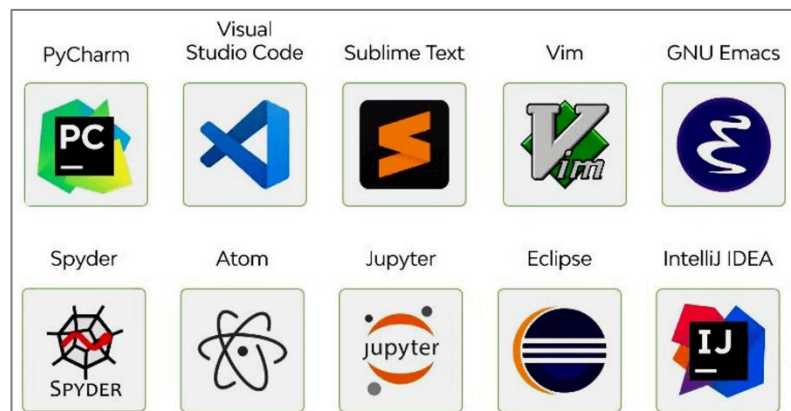
Para trabalhar com uma linguagem de programação é necessária a adoção de um software de trabalho comumente chamado de IDE – abreviação de Integrated Development Environment ou Ambiente Integrado de Desenvolvimento, em português. Um IDE é o programa de computador que os programadores utilizam para escrever, executar, testar, depurar e documentar seus códigos.

Junto com a instalação de Python é instalado também um IDE simples chamado Idle. O Idle pode ser usado para executar instruções de maneira interativa e também para criar, modificar e executar scripts completos. Ele fornece um editor de texto que inclui recursos como destaque de sintaxe, preenchimento automático e recuo inteligente. Ele também possui um depurador básico com pontos de interrupção do código, verificação de conteúdo de variáveis e execução passo-a-passo do código do programa.

Veja o vídeo de demonstração de uso do Idle

Além do Idle, que como dissemos é bastante simples, existem outros IDEs que podem ser usados para escrever programas em linguagem Python e a figura 1.3 exibe alguns (não todos), com seus respectivos Logotipos. Cada um tem seu próprio conjunto de recursos, alguns mais simples e outros mais sofisticados.

Figura 1.3 - IDEs para desenvolver em Python



fonte: o Autor

Todos os IDEs exibidos na figura 1.3 são programas do tipo que o usuário baixa e instala em sua própria máquina. Designamos estes IDEs como sendo off-line, pois uma vez que tenham sido instalados e configurados você poderá usá-los sem a necessidade de manter uma conexão com a internet.

Para escrever os programas deste material foi utilizado o PyCharm. Para ilustrar a instalação e configuração dele foi preparado um vídeo de demonstração.

Veja o vídeo de instalação e configuração do PyCharm

1.3.2 IDEs ON-LINE

Os IDE's on-line representam uma alternativa aos IDEs off-line mencionados acima. Estes IDE's online são práticos pois basta acessar e usar. Não é necessário instalar nada no próprio computador.

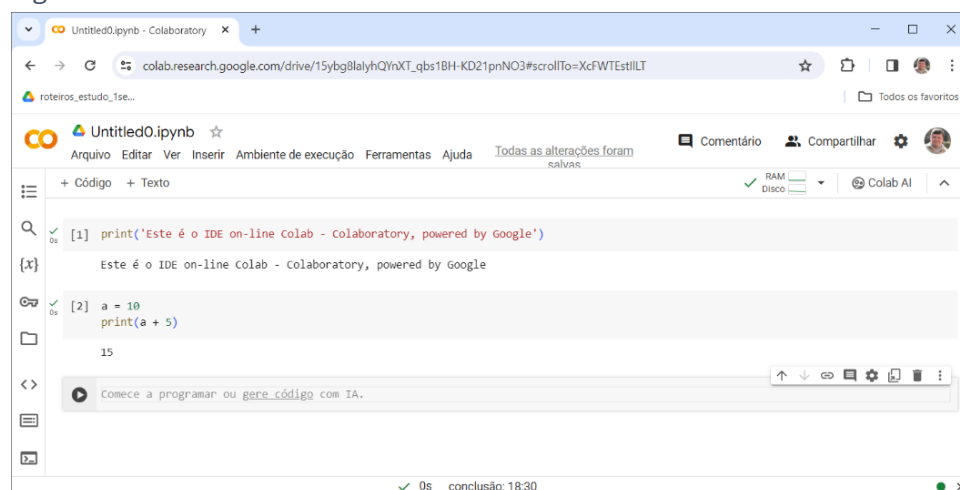
Existem duas vantagens nos ambientes on-line: a primeira é que o acesso e uso pode ser feito a partir de qualquer local e qualquer tipo de aparelho, como computadores, celulares ou tablets. Todo trabalho realizado será armazenado nos servidores on-line (ou “na nuvem” para usar um termo muito corriqueiro atualmente) o que implica que estará disponível para acesso de qualquer local que você queira trabalhar.

A segunda vantagem de usar um IDE on-line é que a máquina do usuário não precisa ser poderosa, pois todo o processamento dos comandos ocorrerá no servidor que hospeda o IDE on-line e tais servidores costumam ser máquinas com grande capacidade de processamento.

Por outro lado, há uma desvantagem a ser considerada. Para usar uma IDE on-line você precisará contar com uma conexão com a internet para utilizá-los.

Um IDE on-line muito utilizado ao redor do mundo é o Colaboratory, carinhosamente conhecido como Colab, desenvolvido e disponibilizado gratuitamente (até certos limites de uso) pelo Google. A figura a seguir mostra a imagem do Colab.

Figura 1.4 - Tela do IDE on-line Colab



fonte: o Autor

Para usar o Colaboratory é obrigatório que o usuário tenha uma conta Google. Caso você não tenha uma, poderá criá-la gratuitamente. Além disso, depois de acessar sua conta Google você precisará ativar o serviço Colaboratory associado a essa conta.

Veja o vídeo sobre como fazer a ativação do serviço Colab vinculado à sua conta Google e como usá-lo



1.4 DOCUMENTAÇÃO E SUPORTE AO PYTHON 3

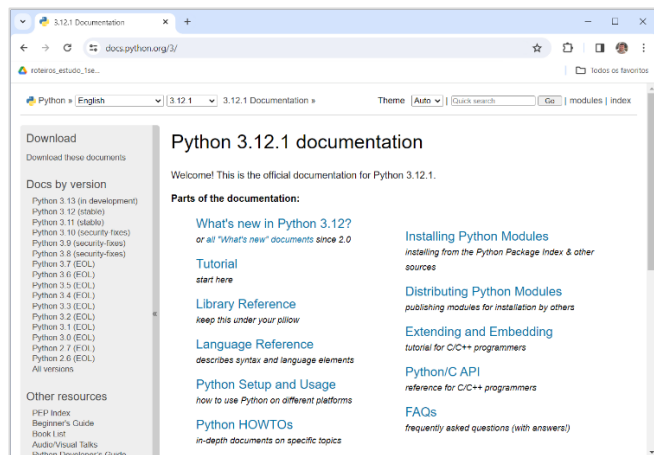
A comunidade Python é muito ativa e como consequência disso muito material de apoio tem sido produzido e disponibilizado aos desenvolvedores interessados em usá-la.

A documentação primária que deve ser a referência para todo programador Python é conhecida como **Python Docs** e está disponível online no endereço <https://docs.python.org/3>

Quando tiver o Python instalado na sua máquina você também terá acesso ao Python Docs. Para acessá-lo basta entrar no IDE Idle e pressionar a tecla F1 (em PCs).

A figura 1.5 mostra a tela inicial do Python Docs, conforme apresentada em janeiro de 2024, época em que este material foi produzido. Neste momento a versão mais recente é a 3.12.1 e esta página reflete este fato. À medida que as versões se sucedem o conteúdo de Python Docs é atualizado de modo que qualquer acesso feito em um momento futuro poderá exibir na tela inicial uma versão posterior a essa atual.

Figura 1.5 - Tela inicial do Python Docs - Documentação oficial e primária da linguagem Python



fonte: <https://docs.python.org/3/> – acesso em 05 jan 2024

Somando-se ao Python Docs há uma vasta disponibilidade de fontes de informação sobre Python, tais como: artigos, vídeos, cursos, fóruns, listas de discussões, blogs, tudo isso acessível online em vários idiomas, inclusive português. Também existe uma grande quantidade de livros publicados sobre o assunto.

Para os brasileiros, a comunidade Python Brasil é excelente recurso em português. Acesse a comunidade em: <http://python.org.br>

Outra referência importante é o conjunto de PEPs (Python Enhancement Proposals). PEP é um documento padronizado utilizado para diversos fins, como: formalizar a divulgação de informações à comunidade; descrever uma nova funcionalidade; apresentação de propostas para novos recursos; coleta de informações sobre problemas; documentação de decisões de projeto adotadas; entre outros.

1.5 REQUISITOS MÍNIMOS PARA RODAR OS EXEMPLOS DESTES MATERIAIS

Programas escritos em linguagem Python não exigem grandes capacidades computacionais. Um computador antigo com processador i3 das primeiras gerações com 2 Gbytes de memória consegue rodar programas Python. Em particular os exemplos deste material podem ser executados em máquinas básicas e de baixo custo. Uma máquina mediana será capaz de executá-los com folga.

Considerando as máquinas que em geral estão disponíveis em 2024, ano em que este material foi produzido, pode-se dizer que qualquer computador a que você tiver acesso executará os exemplos sem problemas.

Além disso, todos os exemplos contidos neste material foram escritos e testados com uso de Python versão 3.12.1 para o sistema operacional Windows. Nestes programas não foi usado nenhum recurso específico da plataforma Windows de modo que se você utiliza outro sistema operacional não encontrará dificuldades e os exemplos devem funcionar corretamente e sem problemas.

1.6 ELEMENTOS FUNDAMENTAIS DE PYTHON

Um programa de computador é um conjunto de instruções que devem ser organizadas de um modo apropriado, na sequência correta e de modo a produzir um resultado esperado. De maneira geral, para escrever um programa de computador, em qualquer linguagem de programação, você sempre vai precisar de duas coisas:

1. algum lugar para guardar dados e
2. comandos para manipular esses dados e produzir um resultado.

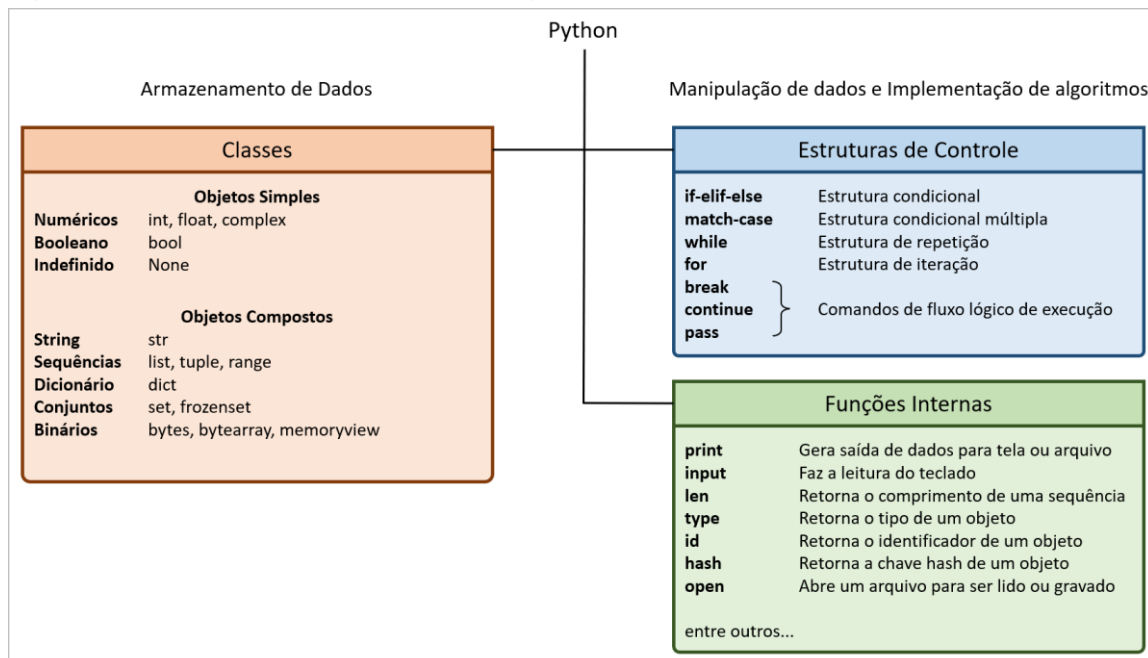
Por exemplo, imagine que você foi a uma papelaria e comprou os itens do quadro a seguir.

Item	Qtde	Preço por unidade	Preço total
Caderno 100 folhas	2	36,90	73,80
Caneta preta	5	6,50	32,50
			106,30

Os dados contidos nas colunas "Item", "Qtde" e "Preço por Unidade" são **dados de entrada**. A coluna azul "Preço total" contém **resultados produzidos** a partir de uma manipulação dos dados de entrada. Você pode escrever um programa em qualquer linguagem de programação para produzir esses resultados e exibí-los na tela. Com esse exemplo fica claro que será necessário que o programa tenha algum elemento onde os dados de entrada e os resultados calculados possam ser armazenados. Também é preciso que existam os comandos que permitam que os cálculos sejam realizados.

A linguagem Python é fundamentada em três categorias de elementos mostrados na figura 1.6.

Figura 1.6 - Elementos fundamentais de Python



fonte: o Autor

No lado esquerdo da figura foi posicionada a categoria "Classes" que contém os elementos que armazenam os dados. No lado direito estão as categorias que permitem a manipulação dos dados, que são as "Estruturas de Controle" e as "Funções Internas" da linguagem Python.

Não se preocupe agora com os elementos que estão listados dentro dos quadros. Eles serão vistos em detalhes adiante e foram colocados aí apenas para que você tenha um resumo para acesso rápido no futuro.

1.7 COMENTÁRIOS

Do que foi dito na seção anterior resulta que um programa de computador é um texto escrito com palavras que representam comandos, funções, classes e objetos. E todos eles tem que ser escritos de modo correto, caso contrário o interpretador da linguagem vai dar erro.

No entanto, há situações em que desejamos escrever textos explicativos no meio do nosso código. Esses textos chamamos de comentários e há o modo certo de escrevê-los de maneira que o interpretador da linguagem os ignore e eles não interfiram na lógica de processamento do programa.

Comentários são algo muito importante no código de um programa e sua inserção é uma prática normal. Toda linguagem de programação tem alguma maneira de permitir que comentários sejam inseridos e em Python não é diferente. Eles são usados para adicionar descrições a partes do seu código, para documentar e descrever os algoritmos implementados.

É comum existirem duas formas de comentários: de uma linha e de múltiplas linhas.

Comentário de uma linha em Python

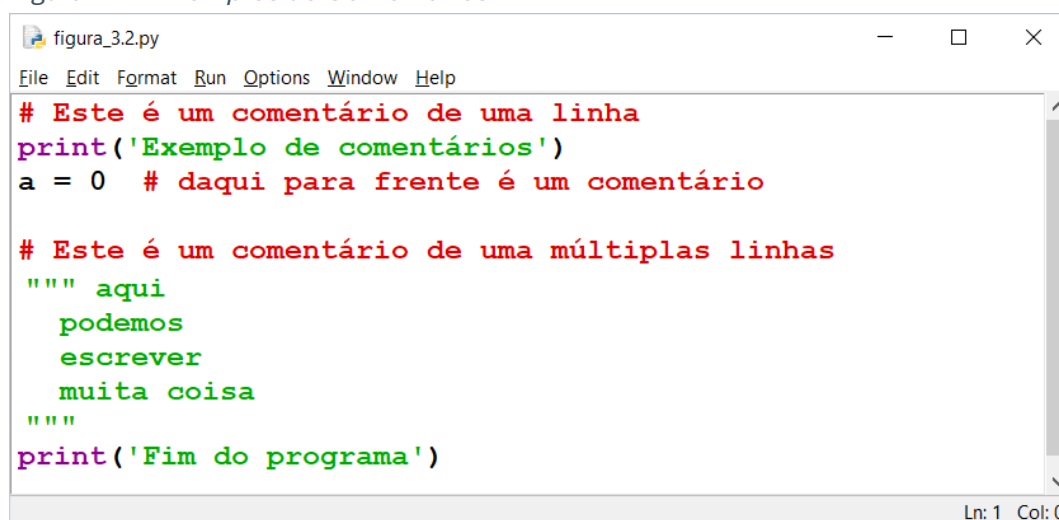
Esta primeira forma usa o caractere cerquilha `#` (popularizado atualmente com o nome *hashtag*) para comentar uma única linha. Não necessariamente esse caractere precisa ser posicionado no início da linha. Quando esse caractere é utilizado, o interpretador ignorará todo o restante da linha até o seu final. Essa forma de comentário já foi usada algumas vezes em exemplo anteriores, como no exemplo 3.7.

Comentário de múltiplas linhas em Python - *docstring*

A segunda forma utiliza três aspas para abrir o bloco de comentário e outras três para fechá-lo. Esta forma é conhecida pelo termo ***docstring***. Todas as linhas que estiverem dentro do bloco serão consideradas como comentário. Podem ser usadas aspas simples ou duplas, porém existe uma recomendação para que preferencialmente sejam usadas aspas duplas em *docstrings* (PEP-0257).

A figura 3.2 exemplifica os dois tipos de comentário.

Figura 1.7 – Exemplos de Comentários



```
figura_3.2.py
File Edit Format Run Options Window Help
# Este é um comentário de uma linha
print('Exemplo de comentários')
a = 0 # daqui para frente é um comentário

# Este é um comentário de uma múltiplas linhas
""" aqui
podemos
escrever
muita coisa
"""
print('Fim do programa')
```

Ln: 1 Col: 0

fonte: o Autor

Lembre-se
Bons profissionais comentam seus códigos.

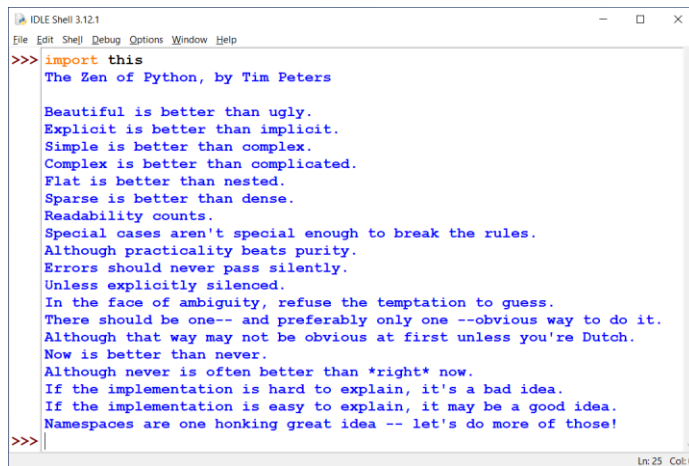
1.8 O ZEN DO PYTHON

O *Zen do Python* é uma coleção de 19 princípios ou aforismos que influenciam a forma de escrever os programas em Python. Quem elaborou essa lista foi o engenheiro de software Tim Peters em 1999 e ela pode ser visualizada quando digitamos:

```
import this
```

Ao fazer isso o resultado é este da figura 1.7.

Figura 1.8 – Zen do Python

A screenshot of the Python IDLE Shell window. The title bar says 'IDLE Shell 3.12.1'. The menu bar includes 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The prompt is '>>>'. The user has entered 'import this'. The output is: 'The Zen of Python, by Tim Peters' followed by 19 lines of text: 'Beautiful is better than ugly.', 'Explicit is better than implicit.', 'Simple is better than complex.', 'Complex is better than complicated.', 'Flat is better than nested.', 'Sparse is better than dense.', 'Readability counts.', 'Special cases aren't special enough to break the rules.', 'Although practicality beats purity.', 'Errors should never pass silently.', 'Unless explicitly silenced.', 'In the face of ambiguity, refuse the temptation to guess.', 'There should be one-- and preferably only one --obvious way to do it.', 'Although that way may not be obvious at first unless you're Dutch.', 'Now is better than never.', 'Although never is often better than *right* now.', 'If the implementation is hard to explain, it's a bad idea.', 'If the implementation is easy to explain, it may be a good idea.', and 'Namespaces are one honking great idea -- let's do more of those!'. The status bar at the bottom right shows 'Ln: 25 Col: 0'.

fonte: Python Software Foundation

O Zen do Python conduz aos 4 princípios que norteiam o desenvolvimento e uso da linguagem

- Legibilidade do código
- Simplicidade
- Modularidade e reutilização de código
- Capacidade de integração

Uma tradução livre para português ficaria assim:

- Bonito é melhor que feio.
- Explícito é melhor que implícito.
- Simples é melhor que complexo.
- Complexo é melhor que complicado.
- Linear é melhor que aninhado.
- Esparsa é melhor que densa.
- Legibilidade conta.
- Casos especiais não são especiais o bastante para quebrar as regras.
- Ainda que praticidade vença a pureza.
- Erros nunca devem passar silenciosamente.
- A menos que sejam explicitamente silenciados.
- Diante da ambiguidade, recuse a tentação de adivinhar.
- Dever haver um — e preferencialmente apenas um — modo óbvio para fazer algo.
- Embora esse modo possa não ser óbvio a princípio a menos que você seja holandês.
- Agora é melhor que nunca.
- Apesar de que nunca normalmente é melhor do que exatamente agora
- Se a implementação é difícil de explicar, é uma má ideia
- Se a implementação é fácil de explicar, pode ser uma boa ideia
- Namespaces são uma grande ideia — vamos fazer mais deles!

Para saber mais sobre o Zen do Python consulte a PEP (Python Enhancement Proposal) 20
<https://peps.python.org/pep-0020/>

Capítulo 2

CLASSES E OBJETOS

2.1 CONCEITOS DE CLASSES E OBJETOS

Conforme mencionado no final do capítulo 1, toda linguagem de programação precisa prever algum meio para armazenar dados. O elemento usado para esse fim é denominado "variável", cujo conteúdo pode ser um número inteiro, um número real ou um texto (sequências de caracteres), entre outros.

Vamos aprofundar essa ideia para em seguida entender como isso é feito em Python.

2.1.1 ARMAZENAMENTO DE DADOS EM PROGRAMAS - VARIÁVEIS

Em programação de computadores uma variável é um elemento da linguagem que ocupa um ou mais bytes na memória do computador. Esse local da memória é capaz de reter, ou seja, armazenar um valor. No programa, a variável é identificada por um nome ou identificador. Desta forma podemos entender que "do ponto de vista" do programador a variável é um nome que contém um valor; e "do ponto de vista" do computador a variável é um endereço de memória que retém um conjunto de bits que representam esse valor.

Está na hora de um exemplo prático para que você perceba que toda essa explicação na verdade representa algo bem simples. Leia todas as linhas do exemplo 2.1 e a observação que está logo após.

Exemplo 2.1



```
qtde = 2
puni = 36.90
ptot = qtde * puni
msg = 'Total ='
print(msg, ptot)
Total = 73.8
```

Observação importante – Explicação sobre a estrutura do exemplo que vale para todos os exemplos contidos neste material. As linhas mais escuras representam o código do programa e as linhas mais claras na sequência representam o resultado que será visível na tela quando o programa for executado.

Na primeira linha quando fazemos `qtde = 2` o valor 2 está sendo atribuído ao nome `qtde`. Assim, dizemos que `qtde` é a variável e 2 é o seu conteúdo. Outras duas variáveis numéricas estão presentes: `puni` para o preço unitário e `ptot` para o preço total que é um resultado calculado pela multiplicação

`qtde * puni`. Uma quarta variável, denominada `msg`, foi criada para conter um texto. Na última linha, a função `print` é usada para exibir na tela o texto contido em `msg` seguido do valor contido em `ptot`.

Ao ser executado, este programa mostrará na tela `Total = 73.8`, conforme mostrado na parte cinza claro da imagem do exemplo.

2.1.2 TIPOS DE DADOS

Ainda fazendo referência ao exemplo 2.1, você pode observar que:

- `qtde`: recebe um número inteiro;
- `puni`: recebe um número real;
- `ptot`: recebe a multiplicação de um número inteiro por um real, por consequência, seu conteúdo será também um número real;
- `msg`: recebe uma sequência de caracteres, então seu conteúdo é um texto.

A expressão "Tipo de Dado" está relacionada à natureza do conteúdo que uma variável pode armazenar. Esse conceito existe em todas as linguagens de programação, porém há algumas variações sobre a forma como isso é implementado na prática em cada uma delas.

Há linguagens em que é obrigatório primeiro definir a variável e ao fazer isso é mandatório indicar qual é o tipo do seu conteúdo. Só depois disso é que o programador pode utilizá-la no programa. Este é o caso das linguagens C e Java. Além disso, nestas linguagens, o tipo da variável não poderá ser alterado ao longo do programa.

Por outro lado, há linguagens, como Python e PHP, que abordam esse conceito de um modo diferente. Nelas não é necessário declarar a variável, nem definir seu tipo previamente. O programador é livre para criá-la no momento que desejar e para fazer isso basta atribuir um conteúdo, como feito no exemplo 2.1. O tipo será automaticamente definido a partir do conteúdo. Nessas linguagens também pode ocorrer que um mesmo identificador tenha um conteúdo numérico inteiro e um pouco adiante no programa esse mesmo identificador passe a conter um texto, ou qualquer outro tipo de dado possível. Este é um aspecto que confere grande flexibilidade ao programador, mas também exige certos cuidados para que o programa não fique incorreto.

Sobre variáveis e tipos de dados pode-se dizer que toda variável possui um nome que a identifica e armazena um conteúdo cujo formato é definido pelo tipo de dado a ela associado.

2.1.3 MODELO DE DADOS DE PYTHON

O Modelo de Dados de Python é um paradigma importante e será apresentado em duas partes. A primeira parte está nesta seção e a segunda está mais adiante neste capítulo.

Nas duas seções anteriores apresentamos os conceitos de variável e tipo de dados. São conceitos gerais que se aplicam a qualquer linguagem de programação.

Agora precisamos formalizar como isso é feito em Python. Cabe ressaltar que **em Python não existem tipos de dados e variáveis** como nas linguagens C ou Pascal.

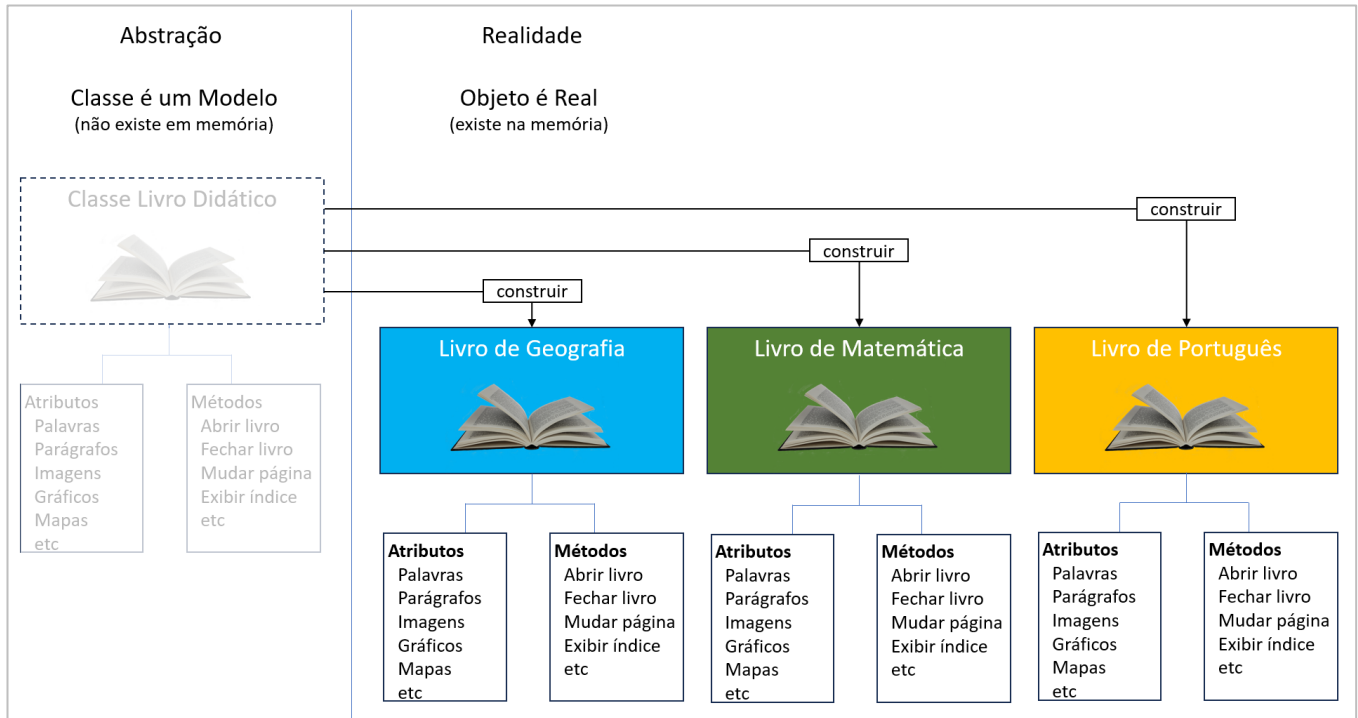
Em Python existem **Classes** e **Objetos**. E esses elementos constituem a base do paradigma de programação conhecido como **Programação Orientada a Objetos** (sigla POO).

E Python é uma linguagem que segue esse paradigma, sendo totalmente orientada a objetos.

Detalhar o conceito de POO neste momento não seria produtivo, pois estamos no início do curso. POO será assunto para o próximo módulo, mas há alguns conceitos básicos que você precisa conhecer para compreender vários tópicos que vamos usar neste módulo.

Então vamos lá. Veja a figura 2.1 onde ilustramos alguns pontos centrais da Orientação a Objetos.

Figura 2.1 - Ilustração sobre Classes e Objetos em POO



fonte: o Autor

Relação entre Classe e Objeto

A figura 2.1 pretende mostrar essa relação. A classe é um modelo a partir do qual os objetos são construídos.

- **Classe** é um modelo que define como um objeto é construído. A classe é codificada no programa e fica à disposição do programador para a ser usada no momento em que for necessário criar um objeto;
- **Objeto** é construído a partir de uma classe. Ele é real, ocupa memória e consome tempo de processamento do computador;

Comparando com linguagens Não-POO pode-se dizer que a classe equivale ao tipo de dado e o objeto equivale à variável.

Pode parecer mera questão de nomenclatura, mas é muito mais que isso. A diferença entre variáveis e objetos está no fato de que as primeiras são um local de armazenamento. Na prática, a variável é apenas uma referência a um endereço de memória do computador onde estão os bytes que representam o valor do dado.

Objetos precisam ser construídos

Ao escrever o código de uma classe o programador está modelando no programa um elemento do mundo real, por exemplo, em um sistema de vendas poderão existir as classes Produto, Comprador,

Vendedor, Impostos, etc. Em cada classe deverão ser previstos os dados que lhe são relevantes e as ações que são realizadas pela classe. Na figura 2.1 foi modelada a classe "Livro Didático", com seus atributos palavras, parágrafos, imagens, etc; e com seus métodos abrir, fechar, etc.

Quando pronta, essa classe fica então disponível para o programador.

No momento de usar a classe o programador deverá construir um ou mais objetos dessa classe. No exemplo da figura 2.1 foram criados três objetos específicos, ou instâncias da classe: um livro didático com conteúdo de geografia, outro com conteúdo de matemática e o terceiro com conteúdo de português.

Classes e Objetos contêm Atributos

Continuando com a figura 2.1, os objetos construídos a partir da classe representam livros didáticos específicos e em sua estrutura vão conter os atributos previstos na classe.

No objeto "livro de geografia" espera-se que o conteúdo trate do assunto geografia e para isto acontecer, basta carregar seus atributos – palavras, parágrafos, imagens, etc – com conteúdos de geografia. O mesmo vale para os outros dois: matemática e português. A estrutura dos atributos é a mesma, mas o conteúdo é diferente. Assim, podemos dizer que o conteúdo dos atributos personaliza o objeto, tornando-o específico para uma determinada aplicação.

Classes e Objetos contêm Métodos

Assim como ocorre para os atributos, também ocorre para os métodos. Cada objeto construído terá todos os métodos previstos na classe.

Os métodos são as funcionalidades de um objeto. Eles atuam sobre os atributos e permitem a execução de ações como entrada e saída de dados (atributos), validações de dados, cálculos, acesso a banco de dados, etc.

Como exemplo suponha que um objeto da classe Trabalhador contenha os atributos "salário bruto" "salário líquido" e "conta corrente". Ele também pode possuir um método para calcular o "salário líquido" a partir do "salário bruto" e outro método para enviar ao banco a informação de depósito do "salário líquido" na "conta corrente".

Estes são os conceitos básicos de POO que você precisa conhecer neste momento.

E com base nesses conceitos podemos dizer que **o Modelo de Dados de Python (Python Data Model) é fundamentado no uso de Classes e Objetos.**

Todo objeto Python possui três aspectos: um **identificador**, uma **classe** e um **conteúdo**.

- O **identificador** é o nome que o objeto tem no programa. Depois de criado esse nome nunca pode ser alterado. Além disso, ele deve ser único, ou seja, não podem existir dois objetos com o mesmo nome dentro de um escopo (o conceito de escopo é algo que ainda precisa ser explicado, mas aqui não é o momento – sobre isso, pense que em um pequeno programa não podem existir dois objetos com o mesmo nome);
- A **classe** é o gabarito usado para criação do objeto, na qual os atributos servem para armazenamento e os métodos servem para executar processamento e realizar operações sobre os atributos. Cada objeto em Python é sempre criado a partir de alguma classe. Assim, um

objeto do tipo número inteiro será da classe `int` e terá associado a si um conjunto de operações adequadas à manipulação de números inteiros; um objeto do tipo lista será da classe `list` e terá as operações adequadas à manipulação de listas; e assim por diante.

- O **conteúdo** do objeto é o que estará armazenado dentro do objeto. Observe que os objetos podem ser simples ou complexos, podendo armazenar apenas um único valor (por exemplo um número inteiro) ou todo um conjunto de valores interrelacionados (por exemplo nome, celular, email e data de nascimento de um amigo seu).

Outro fato é que o identificador precisa ser válido e para isso é preciso seguir as seguintes regras:

- Deve ser escrito com letras maiúsculas (A-Z), minúsculas (a-z), algarismos (0-9) e underline ("_"). Quaisquer outros caracteres não são permitidos;
- Não pode haver espaços em branco;
- Não pode começar com um algarismo;
- Letras maiúsculas e minúsculas são consideradas caracteres distintos (*case-sensitive*);
- Podem ter qualquer quantidade de caracteres, sem limites.

Em um objeto existe a possibilidade de armazenar vários dados interrelacionados, em associação com as operações feitas com esses dados. Isso confere grande poder e flexibilidade aos objetos, tornando-os um elemento de programação bem mais completo quando comparados às simples variáveis.

Veja o vídeo do exemplo 2.2 no qual é usado o ambiente interativo Idle para mostrar esses conceitos. Nele cada objeto é definido e em seguida, usamos a função `type` para mostrar qual é a classe de cada um.

Exemplo 2.2



```
>>> qtde = 2
>>> type(qtde)
<class 'int'>
>>> puni = 36.9
>>> type(puni)
<class 'float'>
>>> ptot = qtde * puni
>>> type(ptot)
<class 'float'>
>>> msg = 'Total ='
>>> type(msg)
<class 'str'>
>>> print(msg, ptot)
Total = 73.8
```

(exemplo interativo feito com IDE Idle)

Considerando o objeto `qtde` deste exemplo veja que estão presentes os três aspectos mencionados: o identificador é `qtde`, a classe é `int` – exibida como `<class 'int'>` – e o conteúdo é o valor 2. O mesmo vale para os outros objetos do exemplo.

Deste ponto em diante neste texto, será usado o termo "classe" ao invés de "tipo" e o termo "objeto" ao invés de "variável", para referência aos elementos relacionados ao armazenamento de dados em programas escritos com Python.

Nas próximas seções apresentamos as classes de objetos pré-existent em Python. Por uma opção didática será feita uma distinção em duas categorias como mostrado no capítulo 1, figura 1.6:

- Objetos simples: compreendem os tipos de dados mais básicos e são chamados de simples pois são indivisíveis. São aqueles cujo valor representa uma única informação;
- Objetos compostos: são constituídos de elementos que podem ser acessados em conjunto ou individualmente (em outras linguagens também são conhecidos como "tipos estruturados").

2.1.4 OBJETOS SIMPLES

Os objetos simples de Python são descritos no quadro 2.1 e exemplificados no exemplo 2.3.

Quadro 2.1 – Objetos simples em Python

Classe	Descrição
int	Contém números inteiros tanto positivos, como negativos (e também zero)
float	Contém números reais positivos e negativos. O ponto deve ser usado como caractere de separação entre as partes inteira e decimal.
complex	Contém números complexos constituídos de parte real e parte imaginária. O valor da parte imaginária é acompanhado do caractere "j". Na matemática usa-se o caractere "i", mas a equipe de Python adotou o "j" seguindo o padrão usado na engenharia.
bool	Contém valores booleanos, ou lógicos. As duas possibilidades são: False para falso e True para verdadeiro, que são escritos com as iniciais maiúsculas.
None	None representa a ausência de valor. Não é algo usado com muita frequência em programação, mas há situações em que a ausência de valor ocorre. Em Python o None (com letra maiúscula) é usado nas situações em que há sua ocorrência. Um caso típico ocorre quando desejamos verificar se um objeto contém valor ou não. Esse objeto poderá ser comparado com None para realizar essa verificação.

Exemplo 2.3



```
>>> n = 12
>>> type(n)
<class 'int'>
>>> f = 7.38
>>> type(f)
<class 'float'>
>>> c = 1.57 + 3.17j
>>> type(c)
<class 'complex'>
>>> b = True
>>> type(b)
<class 'bool'>
>>> x = None
>>> type(x)
<class 'NoneType'>
```

(exemplo interativo feito com IDE Idle)

2.1.5 OBJETOS COMPOSTOS

O contraponto aos objetos simples são os objetos compostos. Eles se caracterizam por serem grupos de elementos, ou seja, um objeto composto contém elementos que são outros objetos.

Os objetos compostos de Python são descritos de forma sucinta no quadro 2.2 para que você saiba da sua existência. Porém, nesta seção não colocaremos exemplos deles, pois precisam ser apresentados em detalhes e com as devidas explicações para que sejam compreendidos. Vários desses objetos serão vistos mais adiante neste módulo do curso. Outros, porém, devido ao seu grau de especialização e aplicabilidade serão vistos nos módulos intermediário e avançado.

Em todos os casos colocamos no quadro a seguir, o link para a documentação oficial do Python que trata do assunto, para que fique como referência rápida facilitando futuras consultas.

Quadro 2.2 – Objetos Compostos em Python

Classe	Descrição
str	<p>Exemplo: <code>S = 'Isto é um Texto!!'</code> ou <code>S = "Isto é um Texto!!"</code></p> <p>A classe <code>str</code> é usada para conter texto, ou seja, uma cadeia de caracteres.</p> <p>Em um programa para definir uma cadeia de caracteres pode-se usar aspas simples ou duplas. Os caracteres são armazenados utilizando-se a codificação Unicode (ao invés de ASCII).</p> <p>Strings são construídos usando aspas. O Python aceita que sejam usadas tanto aspas simples ('), quanto aspas duplas ("), desde que o mesmo tipo de aspas seja usado no início e no final do string.</p> <p>Para saber mais acesse: https://docs.python.org/pt-br/3/library/stdtypes.html#text-sequence-type-str</p>
list	<p>Exemplo: <code>L = [12, 17, 9, 25, 41, 36, 23]</code></p> <p>A classe lista é uma sequência. Listas são usadas para armazenar objetos de qualquer classe. É comum que todos os elementos da lista sejam da mesma classe e quando isso ocorre dizemos que a lista é homogênea. Porém isso não é obrigatório, ou seja, uma lista pode conter elementos classes diferentes e aí dizemos que a lista é heterogênea.</p> <p>Cada elemento da lista pode ser diretamente acessado e alterado através do uso de um índice, sendo que o primeiro elemento terá índice 0 (zero).</p> <p>Listas são construídas com o uso de colchetes [].</p> <p>As listas serão vistas em detalhes no capítulo 7 deste material.</p> <p>Para mais detalhes acesse: https://docs.python.org/pt-br/3/library/stdtypes.html#lists</p>
tuple	<p>Exemplo: <code>T = (23, 18, 99, 48, 56)</code></p> <p>Uma tupla se assemelha a uma lista, porém com a diferença de que ela é apenas para leitura. Ou seja, depois de criada não pode ser alterada. Se houver tentativa de alterar um elemento o interpretador Python gerará uma mensagem de erro.</p> <p>Assim como as listas, as tuplas também podem ser homogêneas ou heterogêneas.</p> <p>Cada elemento da lista pode ser diretamente acessado através do uso de um índice, sendo que o primeiro elemento terá índice 0 (zero).</p> <p>Tuplas são construídas com o uso de parênteses ().</p> <p>As tuplas serão vistas em detalhes no capítulo 7 deste material.</p> <p>Para mais detalhes acesse: https://docs.python.org/pt-br/3/library/stdtypes.html#tuples</p>
range	<p>Range é uma classe usada para criar sequências de números inteiros seguindo as regras de uma progressão aritmética. Para produzir o resultado essa classe deve receber 1, 2 ou 3 parâmetros.</p> <p>Exemplos:</p> <pre>class range(stop)</pre> <p>Se usada deste modo, <code>range</code> produzirá uma sequência iniciando em 0, avançando de 1 em 1 e terminando em <code>stop-1</code></p>

	<pre>class range(start, stop[, step])</pre> <p>Se usada deste modo, range produzirá uma sequência iniciando em <code>start</code>, avançando com o valor <code>step</code> e terminando em <code>stop-1</code>.</p> <p>A classe <code>range</code> é muito usada em conjunto com o comando <code>for</code> para criação de laços de repetição que deve executar um determinado número de vezes previamente conhecido.</p> <p>Range será vista em detalhes no capítulo 7 deste material.</p> <p>Para mais detalhes acesse: https://docs.python.org/pt-br/3/library/stdtypes.html#ranges</p>
dict	<p>Exemplo: <code>D = {'maçã': 9.75, 'laranja': 6.55, 'banana': 5.43}</code></p> <p>A classe <code>dict</code> é conhecida como dicionário ou mapeamento.</p> <p>Trata-se de uma coleção onde cada elemento é definido como um "par chave-valor". A chave identifica o elemento e o valor é o conteúdo. No dicionário a chave é análoga ao índice de uma lista e deve ser usada para acessar e alterar seu valor.</p> <p>Dicionários são construídos com o uso de chaves <code>{ }</code>.</p> <p>Os dicionários serão vistos em detalhes no capítulo 8 deste material.</p> <p>Para mais detalhes acesse: https://docs.python.org/pt-br/3/library/stdtypes.html#mapping-types-dict</p>
set	<p>Exemplo: <code>C = {23, 35, 14, 28, 9, 37}</code></p> <p>A classe <code>set</code> é usada para a criação de conjuntos de elementos obrigatoriamente distintos. A classe <code>set</code> permite inclusão de novos elementos e a exclusão de elementos existentes. Porém, não é possível a alteração de um elemento existente.</p> <p>A classe <code>set</code> suporta as operações definidas na Teoria dos Conjuntos da matemática.</p> <p>A ordem dos elementos dentro do conjunto é estabelecida pelo interpretador Python e o programador não tem controle sobre ela. Um conjunto não pode ser ordenado.</p> <p>É possível livremente converter listas e tuplas em conjuntos e vice-versa. Se uma lista contiver elementos repetidos, ao ser convertida em conjunto os repetidos serão eliminados.</p> <p>Conjuntos são construídos com o uso de chaves <code>{ }</code>. (Opa, Cuidado!)</p> <p>Dicionários também são construídos usando chaves <code>{ }</code>, por isso, é preciso que o programador tenha cuidado para não confundir os dois. Observando a natureza de seus elementos pode-se diferenciar conjuntos de dicionários.</p> <p>Para mais detalhes acesse: https://docs.python.org/pt-br/3/library/stdtypes.html#set-types-set-frozenset</p>
frozenset	<p>A classe <code>frozenset</code> é semelhante à classe <code>set</code> em tudo, com a exceção de que <code>frozenset</code> não permite inclusão ou exclusão de elementos.</p> <p>O link da documentação de <code>frozenset</code> é o mesmo do <code>set</code> (na linha acima deste quadro).</p>
bytes	<p>Estas três classes, <code>bytes</code>, <code>bytearray</code> e <code>memoryview</code>, existem para a realização de tarefas relacionadas à manipulação de dados binários, tais como reproduzir um som mp3, manipular as cores dos pixels de uma imagem, tratar os bytes que trafegam em uma rede, entre muitas outras possibilidades de exemplos.</p> <p>Estas classes não são vistas neste material, pois trata-se de conteúdo avançado e específico.</p> <p>Deixamos aqui o link da documentação para quem desejar conhecer essas classes</p> <p>https://docs.python.org/pt-br/3/library/stdtypes.html#binary-sequence-types-bytes-bytearray-memoryview</p>
bytearray	
memoryview	

2.2 COMANDO DE ATRIBUIÇÃO

O comando de atribuição é um elemento de programação muito simples e necessário existente em todas as linguagens. Na maioria delas é representado pelo caractere de igual: `=`. Na linha a seguir o identificador `a` está recebendo o valor numérico inteiro 10.

```
a = 10
```

A forma geral do comando de atribuição é esta:

```
identificador = expressão
```

Do lado esquerdo deve-se colocar o identificador que é um nome para o objeto.

A expressão do lado direito pode ser uma das seguintes possibilidades:

- um literal (um valor fixo numérico ou texto); ex: `a = 10` ou `m = 'texto'`
- um objeto; ex: `b = a`
- uma fórmula matemática; `c = a + 10`
- uma função; `d = len('exemplo')`
- ou ainda uma expressão com uma combinação dos itens acima.

Em Python esse operador tem um papel adicional, pois é ele que o programador deve usar para criar os objetos que estarão presentes no programa. Nos exemplos 2.1 a 2.3 apresentados anteriormente você pode conferir o uso dele em várias linhas do código. E, mesmo sem a explicação que estamos fazendo agora, certamente entendeu aquelas linhas.

Veja agora o exemplo 2.4 a seguir. Na primeira linha usamos a função `print()` para exibir "a" na tela, porém o resultado foi um erro indicando que o identificador "a" não está definido. Após fazer `a = 10` o mesmo `print()` funcionou corretamente, pois o objeto "a" foi criado na memória do computador com o uso do comando de atribuição.

Exemplo 2.4



```
>>> print(a)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(a)
NameError: name 'a' is not defined

>>> a = 10
>>> print(a)
>>> 10
```

(exemplo interativo feito com IDE Idle)

O comando de atribuição é uma ideia simples e fácil de assimilar. Porém, há mais informações que você precisa saber sobre esse assunto. E são informações importantes para compreensão da segunda parte do Modelo de Dados de Python.

Observe com atenção o exemplo 2.5 em conjunto com a figura 2.2, em especial o uso da função `id()`.

Exemplo 2.5



```
>>> obj1 = 10
>>> type(obj1)
<class 'int'>
>>> id(obj1)
140730012416728
>>> obj2 = 10
>>> id(obj2)
140730012416728
>>> obj3 = 10.0
>>> type(obj3)
<class 'float'>
```

```
>>> id(obj3)
1563128829712
(exemplo interativo feito com IDE Idle)
```

Todo objeto Python é associado a um número inteiro único gerado no momento da criação do objeto e que é mantido constante durante todo o tempo em que o objeto existir. A função interna `id()` é usada para exibi-lo. Sempre que precisarmos nos referir a esse número usaremos o termo `id`.

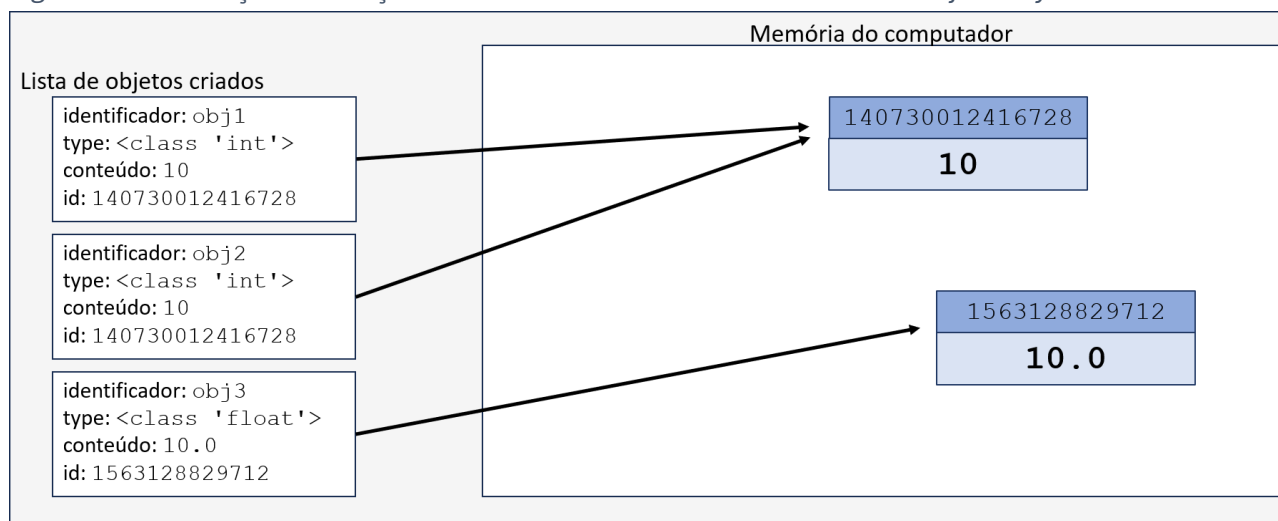
No exemplo 2.5 são criados três objetos nomeados como: `obj1`, `obj2` e `obj3`. Os dois primeiros receberam o valor inteiro 10 e o terceiro recebe o valor real 10.0. A magnitude 10 é a mesma, mas a forma como foi escrito com ponto e uma casa decimal faz toda a diferença.

Se analisarmos esses objetos por tipo de dado (número inteiro – classe `int`) e conteúdo (valor 10) concluímos que `obj1` e `obj2` são idênticos e, em casos assim, a estratégia usada em Python leva à criação de apenas um objeto e os dois identificadores `obj1` e `obj2` serão associados a esse objeto. E com isso, ambos ficam associados ao mesmo `id`.

Por outro lado, o objeto `obj3` é diferente, mesmo com seu valor também sendo de magnitude 10. Como ele foi criado usando-se o valor no formato de número real o Python entende que deve criar um objeto da classe `float`. E assim esse objeto se diferencia dos dois anteriores e estará associado a um `id` diferente.

Essa é a estratégia de gerenciamento de memória adotada em Python.

Figura 2.2 – Ilustração da relação entre o identificador e o número `id` dos objetos Python



fonte: o Autor

Cada vez que o interpretador Python precisar processar um comando de atribuição "=", ele vai avaliar a expressão e determinar a classe e um conteúdo para o objeto, em seguida verificará que se já existe um `id` que satisfaça essa combinação: se existir, ele faz a associação do identificador com o número `id`; se não existir, ele cria um `id` para o novo objeto.

O exemplo 2.6 ilustra uma situação em que três objetos: `x`, `y`, e `z` são criados de diferentes formas, que levam ao mesmo resultado, o valor 50. Ao verificar o `id` dos três objetos constatamos que é o mesmo número associado aos três objetos.

Exemplo 2.6

```
>>> x = 50
>>> y = 12 + 38
>>> z = 100 - 50
>>> id(x)
140730012418008
>>> id(y)
140730012418008
>>> id(z)
140730012418008
>>> print(x, y, z)
50 50 50
>>> z = 90
>>> id(z)
140730012419288
```

(exemplo interativo feito com IDE Idle)

Agora, observe o que acontece com o objeto `z` quando alteramos seu valor para 90. Seu `id` é alterado. Este fato nos leva à próxima seção.

2.3 MODELO DE DADOS DE PYTHON - APROFUNDAMENTO

Observe o que ocorre na Parte 1 do exemplo 2.7 com o `id` do objeto `obj1`. Ao receber o primeiro valor (independe de qual seja o valor) o objeto tem um `id`. Depois quando recebe o segundo valor seu `id` é alterado.

Agora observe o que ocorre na Parte 2 do exemplo. Uma lista foi criada contendo 5 valores. Ao ser criada, a lista ganhou um `id`. Em seguida um dos elementos da lista foi alterado com a atribuição `L[0] = 12` e note que após essa alteração o `id` da lista não mudou.

Exemplo 2.7

```
# Parte 1 - Exemplo usando número inteiro
>>> obj1 = 8
>>> id(obj1)
140730012416664
>>> obj1 = 12
>>> id(obj1)
140730012416792

# Parte 2 - Exemplo usando uma lista
>>> L = [44, 17, 26, 35, 20]
>>> id(L)
1563134844544
>>> L[0] = 12
>>> print(L)
[12, 17, 26, 35, 20]
>>> id(L)
1563134844544
```

(exemplo interativo feito com IDE Idle)

Agora que já verificamos esses fatos relativos ao `id` de objetos podemos nos aprofundar no Modelo de Dados de Python.

Os objetos em Python são classificados em duas categorias: **imutável** e **mutável**.

Você pode escrever seus primeiros programas em Python sem conhecer detalhes sobre esses termos. Porém, este é um conceito muito importante dentro dessa linguagem. Tanto isso é verdade, que basta acessar qualquer um dos links listados no quadro 2.2 e você vai verificar que a página oficial da documentação Python informa para cada classe se ela é imutável ou mutável.

Compreender a distinção entre essas duas categorias de objetos em Python é fundamental para criar um código livre de erros e que também seja eficiente e confiável.

Nesta seção vamos nos concentrar em definir cada categoria e ao longo do restante deste material vamos sempre nos referir a elas quando estivermos nos aprofundando nos elementos da linguagem.

2.3.1 OBJETOS IMUTÁVEIS

Em termos práticos: quando um objeto é imutável, depois de criado, se for preciso trocar seu conteúdo o objeto antigo é descartado e um novo é criado.

Números inteiros, números reais, strings, tuplas entre outros são imutáveis.

A documentação oficial informa que os Objetos Imutáveis são aqueles cujos conteúdos não podem ser alterados após serem criados. Veja o exemplo 2.7. Nesse exemplo o objeto `obj1` foi criado, recebeu um `id`, e foi carregado com um inteiro de valor 8. Como inteiros são imutáveis, ao ser feita a atribuição `obj1 = 12` o objeto antigo foi descartado e um novo foi criado com outro `id`. Isso é ser imutável!

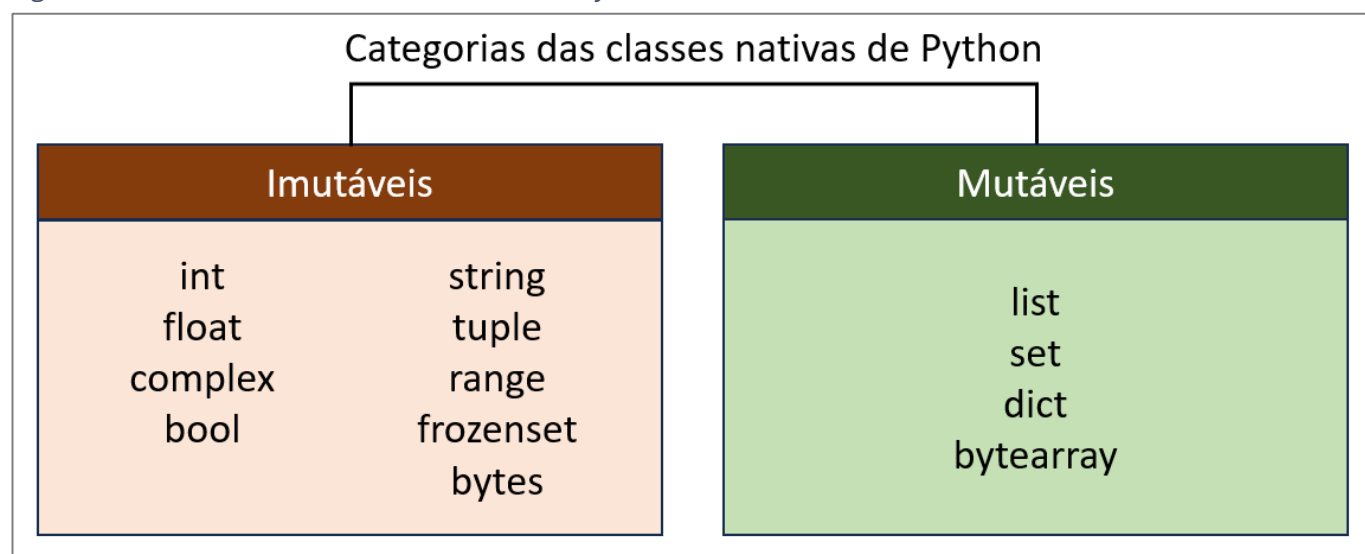
2.3.2 OBJETOS MUTÁVEIS

Os objetos mutáveis são o outro lado dessa ideia. Ou seja, são aqueles cujo conteúdo pode ser alterado após serem criados. O `id` desses objetos se mantém o mesmo durante toda a vida dele enquanto seu conteúdo pode ser livremente alterado sempre que necessário. Isso fica claro na segunda parte do exemplo 2.7 onde uma lista foi criada, teve um elemento alterado e seu `id` permaneceu o mesmo.

Os objetos mutáveis são tipicamente as listas, dicionários e conjuntos.

A figura 2.3 exibe todas as classes de Python categorizadas em imutáveis e mutáveis

Figura 2.3 - Classes Imutáveis e Mutáveis em Python



fonte: o Autor

2.4 ATRIBUIÇÃO MÚLTIPLA

Em Python é possível fazer atribuição múltipla conforme exemplificado na linha abaixo:

```
A = B = 10          # cria dois identificadores que apontam para um objeto int
A = B = 3.5         # cria dois identificadores que apontam para um objeto float
A = B = 'python'    # cria dois identificadores que apontam para um objeto string
```

Como as classes `int`, `float` e `string` são imutáveis, ao encontrar linhas como essas, o interpretador Python criará um objeto em memória e atribuirá seu id aos dois identificadores A e B. Isso vale para todas as classes imutáveis.

Porém, tome cuidado com as classes mutáveis, como listas e dicionários.

```
A = B = []          # cria dois identificadores que apontam para um objeto list
```

Como a classe `list` é mutável e os dois objetos A e B se referem à mesma lista, quando ocorrer alteração em uma delas, essa alteração vai se refletir na outra. Fique atento a isso, mas se você ainda não compreendeu muito bem não se preocupe. Este assunto será mostrado em detalhes na seção 7.2.5.

Outra forma de atribuição múltipla é mostrada a seguir:

```
A, B = 10, 15
X, Y, Z = 12, 7.5, 8.43
```

Neste caso temos dois ou mais identificadores do lado esquerdo e o mesmo número de valores do lado direito. Em comandos assim o Python vai fazer a atribuição conforme a posição, de modo

- na primeira linha: A receberá 10, B receberá 15 e ambos serão inteiros (classe `int`);
- na segunda linha: X receberá 12 e será inteiro; Y e Z receberão 7.5 e 8.43 respectivamente e serão números reais (classe `float`).

Digite essas linhas no Idle e use `print()` e `type()` para ver como ficam cada um dos objetos.

2.5 EXPRESSÕES ARITMÉTICAS

2.5.1 CONCEITOS INICIAIS

As expressões aritméticas existentes na matemática podem ser executadas em todas as linguagens de programação. Essas operações são realizadas sobre valores numéricos; inteiros, reais e complexos.

Em Python, essas expressões são construídas utilizando-se objetos de classes numéricas, operadores aritméticos e funções matemáticas. Uma expressão aritmética é algo como essa linha a seguir:

```
R = A + B
```

onde: A e B são objetos numéricos e R recebe o resultado de sua adição. Nessa expressão, A e B são chamados de operandos e `+` é o operador aritmético de adição.

É possível misturar objetos de diferentes objetos numéricos em uma única expressão. Quando houver uma situação assim, o interpretador Python buscará a melhor maneira de resolvê-la. Havendo, em uma expressão, a mistura de operandos inteiros e reais, o resultado calculado será real. E quando houver inteiros, reais e complexos, o valor resultante será tratado como complexo.

Os operadores aritméticos disponíveis em Python são os indicados no quadro 2.3. Para os exemplos contidos neste quadro foram usados os valores `A = 14` e `B = 5`.

Quadro 2.3 - Operadores Aritméticos

Operação	Operador	Exemplo	Explicação do Exemplo à esquerda	Resultado esperado
Adição	+	$C = A + B$	Soma de A com B	19
Subtração	-	$C = A - B$	Subtrai B de A	9
Multiplicação	*	$C = A * B$	Multiplica A por B	70
Divisão	/	$C = A / B$	Divide A por B, gerando um resultado real	2.8
Divisão inteira	//	$C = A // B$	Divide A por B, gerando um resultado inteiro	2
Resto (módulo)	%	$C = A \% B$	Calcula o resto da divisão de A por B	4
Menos unário	-	$C = -A$	Negativa o valor de A	-14
Potenciação	**	$C = A ** B$	Eleva A por B	537824

As operações indicadas no quadro são mostradas no exemplo 2.8 a seguir.

Exemplo 2.8



```
>>> A = 14
>>> B = 5
>>> C = A + B
>>> print(C)
19
>>> C = A - B
>>> print(C)
9
>>> C = A * B
>>> print(C)
70
>>> C = A / B
>>> print(C)
2.8
>>> C = A // B
>>> print(C)
2
>>> C = A % B
>>> print(C)
4
>>> C = -A
>>> print(C)
-14
>>> C = A ** B
>>> print(C)
537824
```

(exemplo interativo feito com IDE Idle)

Sugestão de exercício: execute todas as operações do exemplo acima com os valores: $A = 14$ e $B = 5$. Obedeça ao esquema a seguir e compare os resultados que você obteve com os resultados esperados indicados no quadro 2.3.

Praticamente todos os operadores apresentados no quadro 2.3 estão disponíveis para serem utilizados com as três classes numéricas definidas na linguagem: int, float e complex.

No entanto, há duas exceções: os operadores de cálculo de divisão de inteiros e cálculo de resto não estão definidos para os objetos complexos.

2.5.2 EXPRESSÕES ARITMÉTICAS COM MÚLTIPLOS OPERANDOS

Em programação, é comum escrever expressões aritméticas envolvendo dois ou mais operadores aritméticos. Em casos assim a ordem de prioridade entre os operadores deve ser observada. O operador de maior prioridade sempre será calculado primeiro. As convenções de prioridade de operações usadas nas linguagens de programação são as mesmas da matemática. Na expressão a seguir, primeiro será calculada a multiplicação entre 2 e A, e ao resultado será adicionado B.

```
R = 2 * A + B
```

Onde for necessário, pode-se alterar a prioridade das operações com o uso de parênteses colocados na expressão de maneira apropriada. Se o desejado para a expressão acima fosse somar A e B primeiro e multiplicar o resultado dessa soma por 2 em seguida, então, a expressão deve ser escrita da forma abaixo.

```
R = 2 * (A + B)
```

Nesta expressão note o uso dos parênteses.

Usando os princípios e convenções estabelecidos na álgebra, juntamente com os operadores aritméticos apresentados no quadro 2.3 pode-se escrever qualquer expressão aritmética em Python.

Exercício Proposto – Execute as expressões sugeridas

```
>>> A = 10
>>> B = 25
>>> C = 7
>>> R = 2 * A + B
>>> print(R)

>>> R = 2 * (A + B)
>>> print(R)

>>> X = (A + B) / (A + C)
>>> print(X)

>>> X = (2 * B - C) / (2 * A)
>>> print(X)

>>> X = -A * 2 + B
>>> print(X)

>>> X = A % B * C
>>> print(X)
```

(exemplo interativo feito com IDE Idle)

2.5.3 COMANDO DE ATRIBUIÇÃO INCREMENTAL

É muito comum em programação que

Uma forma muito frequente de expressão aritmética usada nos algoritmos é aquela em que se toma o conteúdo de um objeto e a ele se soma um certo valor ou outra variável. Uma expressão assim é escrita assim:

```
A = A + 1
```

Para compreender esta expressão primeiro deve-se olhar para seu lado direito: o interpretador irá executar o cálculo $A + 1$ e produzir um resultado. Esse resultado será atribuído ao objeto presente no lado esquerdo da expressão, que é o próprio A. Assim, haverá a substituição do valor original pelo novo valor calculado.

Em Python, nestes casos, pode-se usar a operação de atribuição incremental, que tem a seguinte construção:

```
A += 1
```

Para a atribuição incremental aplicam-se todos os operadores aritméticos previstos em Python. E o valor a ser usado no incremento pode ser qualquer um, tanto literal quanto objetos. Veja o exemplo 2.9.

Exemplo 2.9



```
>>> A = 10
>>> P = 4
>>> A += P      # equivale a A = A + P
>>> print(A)
14
>>> A = 0
>>> A += 15     # equivale a A = A + 15
>>> print(A)
15
>>> A -= 10     # equivale a A = A - 10
>>> print(A)
5
>>> A *= 6      # equivale a A = A * 6
>>> print(A)
30
>>> A /= 4      # equivale a A = A / 4
>>> print(A)
7.5
```

(exemplo interativo feito com IDE Idle)

2.6 FUNÇÕES MATEMÁTICAS

Junto com os operadores aritméticos já apresentados, na linguagem Python pode-se utilizar uma gama muito grande de funções matemáticas. Parte dessas funções estão na biblioteca-padrão e outra parte está em duas bibliotecas externas: "math" contém funções matemáticas que se aplicam a valores numéricos inteiros e reais; "cmath" contém funções matemáticas que se aplicam a números complexos. Ambas nos fornecem uma grande variedade de funções prontas.

A biblioteca-padrão está sempre disponível e não é necessário utilizar nenhum comando específico para usá-la. As bibliotecas math e cmath por sua vez devem ser importadas antes de serem usadas. Veja o exemplo 2.10, no qual foi utilizada a função matemática `sqrt()` capaz de calcular a raiz quadrada de um valor.

Exemplo 2.10



```
>>> from math import sqrt      # importação de função de biblioteca-padrão
>>> X = 49
>>> R = sqrt(X)
>>> print(R)
7.0
```

(exemplo interativo feito com IDE Idle)

O quadro 2.4 algumas funções disponíveis, indicando a qual biblioteca pertencem. Este quadro não esgota todas as possibilidades existentes. Consulte a referência para conhecer todas as funções existentes.

Quadro 2.4 - Funções Python

Função	Descrição	Pertence a
<code>abs(x)</code>	Valor absoluto (módulo) de x	Bib. Padrão
<code>int(x)</code>	Converte x para inteiro eliminando sua parte decimal. O conteúdo de x deve ser real	Bib. Padrão
<code>float(x)</code>	Converte x para número real. O conteúdo de x deve ser inteiro	Bib. Padrão
<code>round(x[, n])</code>	Arredonda x com n dígitos decimais. Se n for omitido, o valor 0 é assumido	Bib. Padrão
<code>trunc(x)</code>	O valor x é truncado, ou seja, a parte decimal é eliminada Na prática, equivale ao <code>int(x)</code>	Bib. math
<code>floor(x)</code>	Retorna o maior inteiro $\leq x$	Bib. math
<code>ceil(x)</code>	Retorna o menor inteiro $\geq x$	Bib. math
<code>sqrt(x)</code>	Calcula a raiz quadrada de x	Bib. math e cmath
<code>exp(x)</code>	Retorna o exponencial de x, ou seja, e^x	Bib. math e cmath
<code>log(x[, base])</code>	Retorna o logaritmo de x na base fornecida. Se a base for omitida, calcula o logaritmo natural	Bib. math e cmath
<code>sin(x)</code>	Retorna o seno do ângulo x radianos	Bib. math e cmath
<code>cos(x)</code>	Retorna o cosseno do ângulo x radianos	Bib. math e cmath
<code>tan(x)</code>	Retorna a tangente do ângulo x radianos	Bib. math e cmath
<code>rect(r, phi)</code>	Converte um número complexo expresso em coordenadas polares para sua representação retangular	Bib. cmath
<code>polar(x)</code>	Retorna a representação do argumento x expresso em coordenadas polares. Retorna uma tupla com o par (r, phi), em que r é o módulo e phi é a fase	Bib. cmath

link para a biblioteca math: <https://docs.python.org/3/library/math.html>

link para a biblioteca cmath: <https://docs.python.org/3/library/cmath.html>

Capítulo 3

COMANDOS DE SAÍDA E ENTRADA

3.1 A FUNÇÃO `print()`

A função `print()` é uma das funções internas de Python 3 e seu propósito é realizar operações de saída de dados. Mas, o que significa esse termo "saída de dados"? Para os iniciantes pode parecer um termo estranho, mas a ideia é simples: saída de dados é exibir na tela ou gravar em um arquivo os conteúdos (valores) dos objetos que estão na memória do computador. Assim, pense que o `print()` é o modo mais básico existente em Python para que o programa "forneça resultados" para quem o está utilizando.

Para realizar suas tarefas a função `print()` tem a estrutura de parâmetros mostrada a seguir.

```
print(*objects, sep=' ', end='\n', file=None, flush=False)
```

onde:

<code>*objects</code>	é o conjunto de objetos cujo conteúdo será exibido;
<code>sep</code>	separador a ser usado quando <code>*objects</code> contiver mais de um objeto. Valor padrão é um branco;
<code>end</code>	contém o caractere a ser enviado para a saída no final. Seu valor padrão é <code>\n</code> (pulo de linha);
<code>file</code>	define o arquivo de saída. Seu valor padrão é <code>None</code> , indicando que a saída é a tela;
<code>flush</code>	se for <code>True</code> indica que o buffer de saída deve ser esvaziado. Seu valor padrão é <code>False</code> .

O único parâmetro obrigatório é `*objects`. Os demais são opcionais e têm valor padrão.

Exemplo 3.1



```
>>> print('Esta é uma mensagem exemplo')           # caso 1
Esta é uma mensagem exemplo
>>> X = 26
>>> print(X)                                       # caso 2
26
>>> Y = 58
>>> print(Y)                                       # caso 2
58
>>> print(X, Y)                                    # caso 3
26 58
>>> print('Valor de X =', X)                       # caso 4
Valor de X = 26
>>> print("Valores: X = {0} e Y = {1}".format(X, Y)) # caso 5
Valores: X = 26 e Y = 58
(exemplo interativo feito com IDE Idle)
```

O `print()` funciona em duas etapas: primeiro os objetos entre os parênteses (que podem ser de qualquer classe) serão convertidos em uma sequência de caracteres de texto (classe `str`); em seguida esse texto é direcionado para o dispositivo do computador.

Caso 1

Neste caso o objeto a ser exibido é um texto literal.

Caso 2

Os prints do caso 2 exibem objetos da classe `int`. O que se quer exibir é o valor numérico contido no objeto de modo que o identificador do objeto é fornecido ao `print()` sem o uso das aspas.

Caso 3

No caso 3, são exibidos simultaneamente os conteúdos de dois objetos `int`. Isso faz com que os valores sejam exibidos na mesma linha separados por um espaço em branco. Esse espaço em branco está pré-configurado no parâmetro `sep`. Em casos assim, é possível alterar esse caractere separador especificando-se um texto alternativo atribuído à `sep`, como mostrado no exemplo 3.2.

Exemplo 3.2



```
>>> A = 12
>>> B = 19
>>> print(A, B, sep="-")           # o separador "-" contém apenas 1 caractere
12-19
>>> print(A, B, sep=" <--> ")     # o separador " <--> " contém 6 caracteres
12 <--> 19
```

(exemplo interativo feito com IDE Idle)

Caso 4

É exibido um texto literal seguido do conteúdo de um objeto, e como o parâmetro `sep` não foi especificado, foi inserido o espaço em branco padrão.

Caso 5

Neste caso é mostrado como produzir uma saída formatada. Esse tipo de saída é muito útil para criar exibições de fácil leitura, pois permite controlar alguns detalhes da exibição dos dados.

É importante você saber que a função `print()` não é a responsável pela formatação. O papel dela se limita a direcionar o texto dos objetos ao dispositivo.

A formatação é realizada pelos métodos disponíveis na classe `str`. No exemplo 3.3 mostramos que o string de saída é criado e carregado no objeto `s` independentemente do `print()`; depois a função `print()` é usada para exibir na tela o conteúdo de `s` já pronto.

Exemplo 3.3

Teste este código no Idle

```
>>> X = 26
>>> Y = 58
>>> s = "Valores: X = {0} e Y = {1}".format(X, Y)
>>> print(s)
Valores: X = 26 e Y = 58
```

(exemplo interativo feito com IDE Idle)

A classe `str` conta três modos de produzir sequências formatadas. Veremos os dois mais populares.

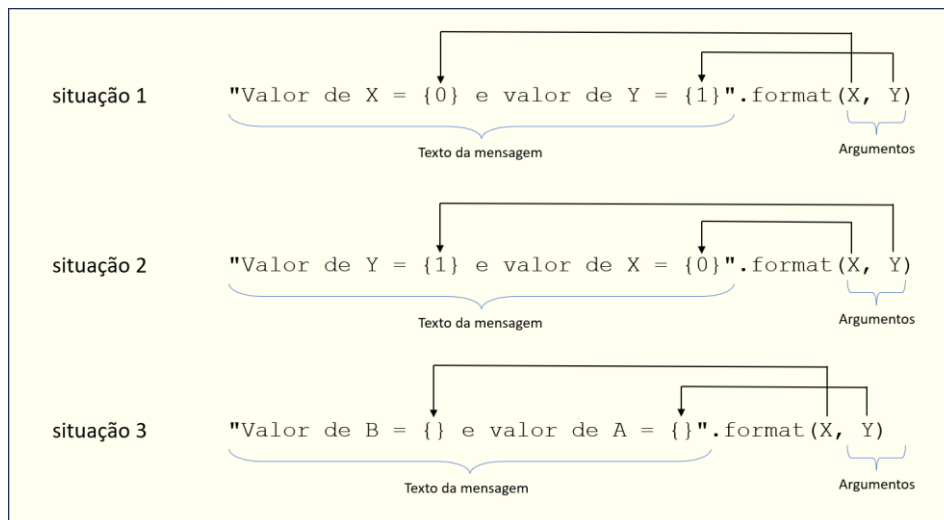
3.1.2 FORMATAÇÃO USANDO O MÉTODO `.format()`

Para produzir uma sequência formatada, o primeiro passo é definir como se quer o texto final, tomando o cuidado de utilizar os marcadores `{0}`, `{1}`, `{2}` etc. nos pontos onde se deseja que apareça o conteúdo dos objetos envolvidos. O texto deve ser seguido do método `.format()`, que conterà como argumentos os objetos que fornecerão os valores que serão usados para substituir os marcadores.

A substituição dos marcadores pelos argumentos é feita seguindo-se o índice numérico, ou seja, nas situações 1 e 2 da figura 3.1 o conteúdo do objeto X substituirá o marcador `{0}` porque X é o primeiro argumento e o conteúdo de Y substituirá o marcador `{1}`, independentemente do local em que esses identificadores estejam posicionados dentro do texto.

Opcionalmente é possível omitir o número dentro das chaves dos marcadores, utilizando apenas `{}`. Neste caso, a associação entre marcador e objeto será feita pela ordem de ocorrência, como mostrado na situação 3 da figura 3.1 (na qual pode-se verificar que é equivalente à situação 1).

Figura 3.1 - Exemplos de saída formatada



fonte: o Autor

Adicionalmente, os marcadores podem receber qualificadores de formatação que definem aspectos de como os dados devem ser formatados. Isso é feito através do uso do caractere `:` (dois pontos) dentro dos marcadores.

São comuns três qualificadores de formatação para os seguintes aspectos:

- Tipo de apresentação: inteiro decimal, binário ou hexadecimal; real; caractere;
- Tamanho da apresentação: quantidade total de caracteres a serem usados e opcionalmente quantidade de casas decimais para os números reais;
- Alinhamento da apresentação: é possível especificar se um dado será alinhado à esquerda, à direita ou centralizado;

O quadro 3.1 mostra típicos casos de uso dos qualificadores de formatação. Sugerimos que você teste cada um deles no Idle para verificar seu funcionamento e absorver a ideia.

Quadro 3.1 - Exemplos de formatação de strings

A = 9 ← Estes objetos A e X são usados nos casos a seguir X = 4.86 As áreas cinzentas na coluna de Resultado realçam o espaço reservado para a inclusão do valor		
Formatação	Resultado em S	Descrição
S = "Dado = {:d}".format(A)	Dado = 9	d – número inteiro, em base 10
S = "Dado = {:5d}".format(A)	Dado = 9	5d – número inteiro ocupando no mínimo 5 caracteres alinhado à direita
S = "Dado = {:f}".format(X)	Dado = 4.860000	f – número real, exibindo o padrão de 6 casas após a vírgula
S = "Dado = {:7.3f}".format(X)	Dado = 4.860	f – número real, ocupando no mínimo 7 caracteres e exibindo 3 casas após a vírgula
S = "Dado = {:.2f}".format(X)	Dado = 4.86	f – número real, exibindo 2 casas após a vírgula. Note que foi usado .2 (para especificar apenas a qtd de casas decimais)
S = "...{:>7d}..." (os pontos servem para marcar início e fim do espaço)	... 9 ...	7d – número inteiro ocupando no mínimo 7 caracteres alinhado à direita (caractere de posição >)
S = "...{:<7d}..."	... 9 ...	7d – número inteiro ocupando no mínimo 7 caracteres alinhado à esquerda (caractere de posição <)
S = "...{: ^7d}..."	... 9 ...	7d – número inteiro ocupando no mínimo 7 caracteres centralizado (caractere de posição ^)

Há outras opções, mas não convém nos estender muito mais, pois são muitos os detalhes e este texto ficaria muito extenso. Partindo dos princípios gerais exemplificados no quadro 3.1, você pode expandir seu conhecimento dos detalhes a partir da exploração da documentação sobre este assunto que está disponível no link a seguir:

Disponível em <https://docs.python.org/pt-br/3/library/string.html#formatspec>

3.1.3 FORMATAÇÃO USANDO f-string

O segundo modo que vamos apresentar é chamado de f-string (uma abreviação para o nome oficial "formatted string literals"). Este modo foi introduzido na versão 3.6 se Python e rapidamente se tornou muito popular pois faz o mesmo que o método `.format()`, porém com uma forma de escrita mais clara e compacta. Veja no exemplo 3.4 as formas usadas para a criação dos objetos `s1` e `s2`. Para `s1` foi usado o método `.format()` visto acima. Para `s2` foi usado o f-string obtendo-se o mesmo resultado.

Exemplo 3.4

```
>>> A = 14
>>> B = 32
>>> s1 = "Valores: A é {} e B é {}".format(A, B)
>>> s2 = f"Valores: A é {A} e B é {B}"
>>> print(s1)
Valores: A é 14 e B é 32
>>> print(s2)
Valores: A é 14 e B é 32
(exemplo interativo feito com IDE Idle)
```



Um *f-string* começa com uma das letras "f" ou "F" e dentro do marcador {} é colocado o objeto que se quer exibir naquela posição.

```
s2 = f"Valores: A é {A} e B é {B}"
```

Todas as demais funcionalidades relativas aos qualificadores de formatação vistas na seção 3.1.2 também se aplicam aos *f-strings*. O quadro 3.2 mostra a correspondência para cada um exemplos.

Formatação com método .format()	Formatação com f-string
S = "Dado = {:d}".format(A)	S = f"Dado = {A:d}"
S = "Dado = {:5d}".format(A)	S = f"Dado = {A:5d}"
S = "Dado = {:f}".format(X)	S = f"Dado = {A:f}"
S = "Dado = {:7.3f}".format(X)	S = f"Dado = {X:7.3f}"
S = "Dado = {:.2f}".format(X)	S = f"Dado = {X:.2f}"
S = "..{:>7d}..".format(A)	S = f"..{A:>7d}.."
S = "..{:<7d}..".format(A)	S = f"..{A:<7d}.."
S = "..{: ^7d}..".format(A)	S = f"..{A:^7d}.."

Existe um terceiro e antigo modo de formatação de strings em Python. Este modo é similar ao usado no comando `printf` da linguagem C e usa o operador de interpolação de string representado pelo caractere %.

Ele é mantido para manter a compatibilidade com softwares que o utilizam.

Não vamos apresentá-lo neste texto e se você tiver interesse em conhecê-lo, acesse este link:

<https://docs.python.org/3/library/stdtypes.html#old-string-formatting>

3.1.4 O QUE É O \n ?

Ao usar a função `print()` é muito comum nos depararmos com a dupla de caracteres `\n`. Isso pode ser encontrado em várias linguagens como C, Java e Python. Na verdade, porém, não se trata de uma dupla de caracteres: o `\n` é uma "sequência de escape" (*escape sequence*). Sequências de escape consistem em uma barra invertida (`\`) seguida de uma letra ou dígitos e representam determinadas ações, como no caso do `\n` que representa a ação de "pulo de linha" ou "avanço de linha".

Sempre que usado junto com `print()` ele faz com que o cursor de tela pule para a linha de baixo. Veja a seguir algumas possibilidades:

```
print('\n')          # pula uma linha na tela
print('um\n dois')   # escreve 'um' em uma linha, pula a linha, escreve 'dois' embaixo
print('\n\n\n')      # pula três linhas
print('\n\nAlgo')    # pula duas linhas e escreve 'Algo'
```

Para funcionar corretamente o `\n` deve estar dentro das aspas

Para saber mais sobre sequências de escape veja esta página da Microsoft
<https://learn.microsoft.com/pt-br/cpp/c-language/escape-sequences>

3.2 A FUNÇÃO `input()`

É preciso que exista alguma forma de permitir que dados sejam digitados no teclado do computador e inseridos em objetos do programa. Toda linguagem de programação tem algum recurso para isso.

Em Python 3 quem realiza essa tarefa é a função `input()`. Para usá-lo basta atribuir seu retorno a algum identificador como mostrado no exemplo 3.5. Opcionalmente pode-se passar um parâmetro que será usado como uma mensagem que indique o que deve ser digitado.

Exemplo 3.5



```
>>> A = input()
    Digitei isto                # o input vai esperar que algo seja digitado
>>> print(A)
    Digitei isto

>>> B = input('Digite algo: ')
    Digite algo: Digitei isto  # o input vai esperar que algo seja digitado
>>> print(B)
    Digitei isto

(exemplo interativo feito com IDE Idle)
```

Quando a função `input()` é executada o computador aguarda que o usuário digite o que precisar seguido de "Enter". Quando a tecla "Enter" é pressionada tudo o que foi digitado é carregado no objeto que recebe o retorno da função.

No exemplo a primeira leitura carrega o objeto `A` sem mostrar qualquer mensagem na tela. Já na segunda leitura é apresentada a mensagem "Digite algo: " indicando ao usuário o que deve ser feito e no retorno dessa função o objeto `B` é carregado.

O usuário pode digitar o que quiser e a leitura sempre resulta em uma cadeia de caracteres (string) carregada no objeto de destino. Se forem digitados apenas algarismos, ainda assim a leitura resulta em uma cadeia de caracteres. Isso pode ser constatado no exemplo 3.6, no qual o objeto `N` aparentemente recebe um número inteiro e `F` aparentemente recebe um número real. Porém, ao utilizar o comando `type`, verifica-se que ambos são do tipo da classe `str`, portanto um string.

Exemplo 3.6



```
>>> N = input("Digite um inteiro: ")
    Digite um inteiro: 37
>>> print(N)
    37
>>> type(N)
    <class 'str'>

>>> F = input("Digite um real: ")
    Digite um real: 31.47
>>> print(F)
    31.47
>>> type(F)
    <class 'str'>

(exemplo interativo feito com IDE Idle)
```

No exemplo pode-se verificar que, de fato, a função `input()` retorna exclusivamente cadeias de caracteres. Agora a questão é: como fazer caso precisemos ler números inteiros ou reais?

A resposta para isso são as funções de conversão de tipo apresentadas na próxima seção.

3.2.1 FUNÇÕES DE CONVERSÃO DE OBJETOS DE CLASSES SIMPLES

Estas funções permitem realizar a conversão entre tipos de dados simples, conforme indicado no Quadro 3.2

Quadro 3.2 - Funções de conversão

Função	Descrição
<code>int(objeto, base)</code>	Converte o objeto para um número inteiro, se for possível. Se base não for fornecida será usada a base 10. As bases permitidas variam de 2 a 36. Caso não seja possível fazer a conversão é gerado um erro. Para saber mais: https://docs.python.org/pt-br/3/library/functions.html#int
<code>float(objeto)</code>	Converte o objeto para um número real, se for possível. Caso não seja possível, gera um erro. Para saber mais: https://docs.python.org/pt-br/3/library/functions.html#float
<code>complex(objeto)</code>	Converte o objeto para um número complexo, se for possível. Caso não seja possível, gera um erro. Para saber mais: https://docs.python.org/pt-br/3/library/functions.html#complex
<code>bool(objeto)</code>	Converte o objeto para um booleano, se for possível. Caso não seja possível, gera um erro.
<code>str(objeto)</code>	Converte o objeto para string. Para saber mais: https://docs.python.org/pt-br/3/library/stdtypes.html#str

O exemplo 3.7 mostra diversos casos de conversão de tipo. Em cada linha colocamos a explicação do que está sendo feito e convidamos você a testar todas as linhas do exemplo no Idle.

Exemplo 3.7

Teste este código no Idle

```
>>> x = '19'           # x recebe o string '19'
>>> type(x)           # conferindo a classe de x
<class 'str'>         # x é da classe str
>>> a = int(x)         # a recebe o retorno da conversão de x para inteiro
>>> type(a)           # conferindo a classe de a
<class 'int'>         # a é da classe int
>>> x = '3.75'         # x recebe o string '3.75'
>>> type(x)           # conferindo a classe de x
<class 'str'>         # x é da classe str
>>> r = float(x)       # r recebe o retorno da conversão de x para real
>>> type(r)           # conferindo a classe de r
<class 'float'>      # r é da classe float
>>> b = a + r          # soma um int com um float, produzindo um float em b
>>> print(b)
22.75
>>> x = str(b)        # converte o float em b para string
>>> print(x)
22.75
>>> type(x)           # conferindo a classe de x
<class 'str'>         # x é da classe str
```

(exemplo interativo feito com IDE Idle)

3.2.2 USO CONJUNTO DO INPUT COM FUNÇÕES DE CONVERSÃO

É muito comum que nos programas seja necessário ler números inteiros e reais. Como a função `input()` sempre lê strings, será necessário executar uma conversão depois da leitura, ou seja, pode-se dizer que a leitura de valores numéricos, inteiros, reais ou complexos, é uma operação de duas etapas.

As duas linhas a seguir mostram como isso é feito.

```
A = input('Digite A: ') # lê o teclado e carrega A com um string
A = int(A)              # converte A para inteiro e coloca esse inteiro em A
```

A segunda linha merece melhor atenção. Note que o objeto A está presente tanto no lado esquerdo como no direito do comando. Em situações assim sempre olhe primeiro para o lado direito e entenda o que ele faz: neste caso string do objeto A está sendo convertido para número inteiro com o uso da função `int()`. Depois, você olha para o lado esquerdo e verifica que o objeto A está recebendo o resultado da conversão. Como objetos da classe `str` são imutáveis o objeto anterior será excluído e um novo objeto A será criado, desta vez baseado na classe `int` e receberá o valor convertido.

Uma vez entendido esse processo podemos dar o próximo passo e escrever o mesmo código assim:

```
A = int(input('Digite A: ')) # lê o teclado, converte para int e carrega A com ele
```

Agora as duas operações ainda estão presentes, porém escritas em uma única linha e tornando o código mais compacto. Esta forma é a mais usada nos programas Python.

O mesmo pode ser feito para a leitura de números reais e complexos.

Exercício Proposto 3.1

Enunciado: Escreva um programa que leia os nomes de três pessoas de uma família: mãe, pai e criança. O programa deve exibir na tela a mensagem.

"Os adultos {mãe} e {pai} são os responsáveis por {criança}"

Faça de dois modos: com o método `.format()` e com f-string

Exercício Proposto 3.2

Enunciado: Escreva um programa que leia um texto e mostre na tela o texto e a quantidade de caracteres que ele contém, usando a seguinte mensagem:

"O texto {AquiColoqueOTexto} contém {Quantidade} caracteres"

Faça de dois modos: com o método `.format()` e com f-string

Dica Use a função `len()`

Exercício Proposto 3.3

Enunciado: Escreva um programa que leia três números reais em objetos denominados A, B e C. O programa deve calcular e mostrar na tela os resultados das fórmulas a seguir, usando 3 casas decimais.

$$R1 = A + B + C \qquad R2 = A \cdot B \cdot C \qquad R3 = 2 \cdot (A + B) - C$$

$$R4 = \frac{A + B + C}{3} \qquad R5 = \frac{2B + 3C}{5A} \qquad R6 = A^2 + B^2$$

Para testar seu programa considere: A = 22.65 B = -39.1 C = 18.115

R1 = 1.665	R2 = -16042.916	R3 = -51.015	R4 = 0.555	R5 = -0.211	R6 = 2041.832
------------	-----------------	--------------	------------	-------------	---------------

Exercício Proposto 3.4

Enunciado: Escreva um programa que leia um número real e mostre na tela os valores de 25%, 50%, 75% do valor lido usando o formato com 3 casas decimais mostrado abaixo:

Exemplo Valor lido: 136.7

Exibir 25% -> 34.175 - 50% -> 68.350 - 75% -> 102.525

Exercício Proposto 3.5

Enunciado: Escreva um programa que leia um número inteiro que representa uma quantidade de tempo em segundos. Calcule e mostre na tela a quantidade de horas, minutos e segundos.

Exemplos:

Entrada (segundos)	Saída
1	0 hora(s), 0 minuto(s), 1 segundo(s)
38	0 hora(s), 0 minuto(s), 38 segundo(s)
746	0 hora(s), 12 minuto(s), 26 segundo(s)
4578	1 hora(s), 16 minuto(s), 18 segundo(s)
73551	20 hora(s), 25 minuto(s), 51 segundo(s)

Dicas: Leve em consideração que 1 hora tem 3600 segundos e 1 minuto tem 60 segundos.

Use os operadores de divisão de inteiros (//) e resto (%).

Exercício Proposto 3.6

Enunciado: Uma empresa comercial trabalha com 3 vendedores externos e os remunera com R\$ 1200,00 fixos mais comissão de 6% sobre o valor total vendido no mês. Escreva um programa que leia o nome e o total vendido pelos 3 vendedores, calcule e exiba na tela a mensagem de saída conforme o exemplo a seguir. Exiba os valores numéricos com duas casas decimais.

Exemplo vendedor José Carlos Santos vendeu R\$ 43759.35 e faz jus a uma comissão de R\$ 3825.56

de saída: vendedor Manoel Guimarães vendeu R\$ 61417.81 e faz jus a uma comissão de R\$ 4885.07

vendedor Plínio Pereira vendeu R\$ 39336.87 e faz jus a uma comissão de R\$ 3560.21

Exercício Proposto 3.7

Enunciado: Quando uma pessoa ou uma empresa realiza um investimento espera-se um retorno positivo (lucro), embora também possa ocorrer um retorno negativo (prejuízo). Uma forma inicial de avaliar o retorno é conhecida com Retorno sobre Investimento (ou ROI, uma sigla em inglês). Cálculo do ROI:

$$ROI = \frac{Receita - (Custos + Investimento)}{Custos + Investimento} \cdot 100\%$$

Escreva um programa que leia 3 dados de entrada reais: Investimento, Custos e Receita, calcule o ROI usando a fórmula acima e exiba o resultado com uma casa decimal no formato mostrado abaixo.

Exemplo: Investimento = 2300.00 – Custos = 345.73 – Receita = 2712,17

Saída: ROI = 2.5%

Outros testes:

Investimento	Custos	Receita	ROI
22500.00	535.83	25419.61	10.3%
15000.00	419.35	14403.44	-6.6%
18000.00	837.40	19132.28	1.6%

Capítulo 4

COMANDO CONDICIONAL

Este capítulo e os próximos dois tratam de comandos da linguagem Python que fazem o controle do fluxo de execução de um programa. Por controle de fluxo em um programa entende-se a ordem lógica de execução dos comandos que o compõe. Isso envolve basicamente três aspectos:

- Desvios na ordem de execução do código em função de certas condições que possam ocorrer. Este tópico será visto neste capítulo;
- Repetição de trechos do programa por um determinado número de vezes, necessária para que um algoritmo possa ser, de fato, implementado. Este tópico será visto no capítulo 5;
- Tratamento de situações de erro que possam ocorrer, também chamado de Tratamento de Exceções. Este tópico será visto no capítulo 6.

4.1 CONCEITO GERAL DE UM COMANDO CONDICIONAL

É muito frequente a necessidade de tomada de decisão em um programa de computador, assim como acontece em situações comuns, por exemplo: "se estiver frio coloque use o casaco, senão tire o casaco".

4.1.1 CONCEITO GERAL SOBRE O COMANDO CONDICIONAL

Em programação essas decisões são baseadas em valores contidos em objetos. Por exemplo, considere uma situação que envolva dois objetos da classe `int` `A` e `B` previamente carregados. Caso seja necessário calcular a divisão de `A` por `B` e o conteúdo do objeto `B` for zero, ocorrerá um erro, como pode ser visto no exemplo 4.1. O motivo do erro é que divisões por zero não são permitidas.

Exemplo 4.1



```
A = 10
B = 0
R = A / B
print(R)

Traceback (most recent call last):
  File "D:\exemplos\exemplo_4.1.py", line 3, in <module>
    R = A / B
    ~~~~
ZeroDivisionError: division by zero
```

Situações de erro assim são indesejáveis e é preciso tomar o cuidado de se evitá-las. Uma das formas (não a única) de se conseguir isso é usar o Comando Condicional: `if-else`. Outra forma será vista no capítulo 6.

Ao utilizar o comando condicional será necessário formular uma condição cujo resultado será uma de duas possibilidades: **falso** ou **verdadeiro**. Em função desse resultado o programa seguirá apenas um de dois possíveis caminhos distintos. A ideia básica é implementar um código que reflita esta frase: "se B for igual a zero, então apresente a mensagem 'Não é possível calcular a divisão', senão (ou seja, B é diferente de zero) calcule e apresente na tela A / B.

Em Python um programa completo capaz de implementar essa ideia é exibido no exemplo 4.2.

Exemplo 4.2



```
A = int(input('Digite A: ')) # linha 1
B = int(input('Digite B: ')) # linha 2
if B == 0: # linha 3
    print('Não é possível calcular a divisão') # linha 4
else: # linha 5
    R = A / B # linha 6
    print(R) # linha 7
```

primeira execução

```
Digite A: 26
Digite B: 0
Não é possível calcular a divisão
```

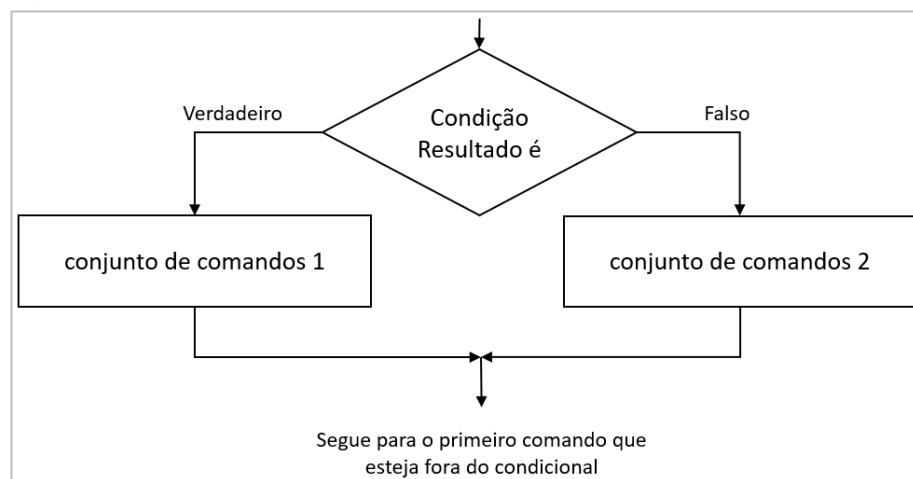
segunda execução

```
Digite A: 26
Digite B: 4
6.5
```

Na primeira execução deste exemplo, foram fornecidos os valores: A = 26 e B = 0 e o resultado foi a exibição da mensagem avisando que não é possível calcular a divisão. Na segunda execução, foram fornecidos os valores: A = 26 e B = 4 e o resultado foi a exibição do valor 6,5 que equivale a 26 dividido por 4.

Este é o conceito básico do Comando Condicional e que é ilustrado na figura 4.1.

Figura 4.1 - Conceito do Comando Condicional



fonte: o Autor

O próximo passo é avançar em direção aos detalhes que ainda não foram mencionados.

4.1.2 ENTENDENDO OS DETALHES DO EXEMPLO 4.2

As linhas do exemplo foram numeradas para facilitar este detalhamento. Nas duas primeiras, 1 e 2, são usadas as funções `input()` e `int()` em conjunto para a leitura dos objetos A e B.

Na linha 3 o comando condicional `if` contém a condição `B == 0`. Nesta condição a pergunta que se está fazendo é se o conteúdo de B é igual a zero. Para isso foi usado o operador relacional `'=='` que avalia se B que está do lado esquerdo contém um valor igual a zero, que está do lado direito. Todos os operadores relacionais são apresentados logo mais, na seção 4.1.4.

Esta condição será avaliada pelo processador e um resultado será gerado. Esse resultado pode ser falso ou verdadeiro. Caso seja verdadeiro, o programa seguirá para a linha 4 e executará a função `print()`. Caso seja falso, o programa vai ignorar a linha 4 e seguirá para a execução das linhas 6 e 7, que estão subordinadas ao `else` (senão) da linha 5.

Note que nas linhas 3 e 5 há um caractere `:` (dois pontos) no final da linha. Em Python é obrigatória a colocação desse caractere no comando tanto no `if`, como no `else`, pois é através dele que o interpretador Python identifica o término do cabeçalho do comando e o início do código subordinado.

Essa relação de subordinação é muito importante na lógica do algoritmo. Neste exemplo a linha 4 está subordinada ao `if` da linha 3 e as linhas 6 e 7 estão subordinadas ao `else` da linha 5.

4.1.3 INDENTAÇÃO

Como dito, a relação de subordinação de um trecho de código a um determinado comando é muito importante na lógica de um programa.

Em programação de computadores esse termo indentação se aplica ao uso de espaços em branco à esquerda de uma linha de código com o objetivo de realçar uma relação de subordinação, definindo assim uma estrutura legível para o programa.

Na maioria das linguagens essa indentação é opcional e cabe ao programador decidir se vai ou não usá-la – e é preciso que fique claro que todos os bons programadores usam, porque torna o código mais legível e fácil de compreender.

No Python a indentação é obrigatória. No exemplo 4.2 note que o `if` e o `else` estão no mesmo alinhamento e seus comandos subordinados estão avançados para a direita indicando uma relação de subordinação. Generalizando, em Python, todo conjunto de comandos subordinados deve estar indentado em relação ao seu comando proprietário. Isso vale para `if-else`, `while`, `for`, `try`, `def` e qualquer outro elemento de Python em que exista a relação de subordinação.

4.2 CONDIÇÕES SIMPLES

O pilar do comando condicional é a condição que será avaliada e cujo resultado pode ser falso ou verdadeiro. Existem condições simples e compostas e nesta seção vamos nos concentrar nas primeiras.

A condição `B == 0` usada no exemplo 4.2 é uma condição simples, pois ela tem a seguinte construção:

```
<expressão 1> <operador> <expressão 2>
```

onde as **expressões 1 e 2** podem ser uma dessas possibilidades:

- um literal (geralmente número ou texto)
- um objeto
- uma fórmula (expressão aritmética)
- uma chamada de função

O **operador** é um dos seis operadores relacionais exibidos no Quadro 4.1. No caso dos operadores que contém dois caracteres, não é permitido haver espaço em branco entre eles.

Quadro 4.1 - Operadores Relacionais

Operador	Como se lê	Interpretação
<code>==</code>	Igual a	<code>X == Y</code> : retorna True se X e Y são iguais, caso contrário retorna False
<code>!=</code>	Diferente de	<code>X != Y</code> : retorna True se X e Y são diferentes, caso contrário retorna False
<code><</code>	Menor que	<code>X < Y</code> : retorna True se X for menor que Y, caso contrário retorna False
<code><=</code>	Menor ou igual a	<code>X <= Y</code> : retorna True se X for menor ou igual a Y, caso contrário retorna False
<code>></code>	Maior que	<code>X > Y</code> : retorna True se X for maior que Y, caso contrário retorna False
<code>>=</code>	Maior ou igual a	<code>X >= Y</code> : retorna True se X for maior ou igual a Y, caso contrário retorna False

O exemplo 4.3 ilustra diversas condições simples e seus resultados. Para condição é colocado um comentário indicando os elementos presentes: literal, objeto, fórmula ou função

Exemplo 4.3

```
>>> A = 10
>>> B = 50
>>> A > 0           # Comparação entre objeto e literal
True               # o resultado é True porque A é maior que zero
>>> B <= 0          # Comparação entre objeto e literal
False              # o resultado é False porque B não é maior ou igual a zero
>>> A >= B          # Comparação entre dois objetos
False              # o resultado é False porque A não é maior ou igual a B
>>> 5 * A == B      # Comparação entre fórmula e objeto
True               # o resultado é True porque 5 vezes A é igual a B
>>> A >= pow(B, 0.5) # Comparação entre objeto e função (raiz quadrada de B)
True               # o resultado é True porque A é maior ou igual à raiz de B
(exemplo interativo feito com IDE Idle)
```

Um detalhe sobre a última condição do exemplo. Nela foi usada a função de exponenciação `pow(base, exp)`. Essa função recebe dois parâmetros `base` e `exp` e calcula $base^{exp}$ (base elevado a exp). Como foi usado o valor 0.5 para `exp` isso equivale a calcular a raiz quadrada.

4.3 NEGAÇÕES E CONDIÇÕES COMPOSTAS

Muitas vezes é preciso negar uma condição simples ou combinar duas ou mais condições simples em uma condição composta. Para isso são usados os **operadores lógicos**. Existem três operadores lógicos em Python:

Quadro 4.2 – Operadores Lógicos disponíveis em Python

Operador	O que ele faz	Descrição
not	Negação	Nega a condição à qual é aplicado
and	Conjunção operação lógica E	Resulta verdadeiro se forem verdadeiras as duas condições às quais é aplicado
or	Disjunção operação lógica OU	Resulta verdadeiro se for verdadeira pelo menos uma das duas condições às quais é aplicado

Cada um desses operadores é regido por uma Tabela Verdade que define seus resultados em função das entradas. A figura 4.2 exemplifica o uso do operador lógico not.

Figura 4.2 – Tabela verdade da operação not

Tabela verdade do operador not Definição: O resultado not C1 é o oposto de C1	
Condição C1	not C1
False	True
True	False

fonte: o Autor

```

IDLE Shell 3.12.1
File Edit Shell Debug Options Window Help
>>> A = 0
>>> A > 0
False
>>> not A > 0
True
Ln: 8 Col: 0

```

A figura 4.3 define e exemplifica o uso do operador lógico and.

Figura 4.3 – Tabela verdade da operação and

Tabela verdade do operador and Definição: O resultado será verdadeiro se as duas condições C1 e C2 forem verdadeiras		
Condição C1	Condição C2	C1 and C2
False	False	False
False	True	False
True	False	False
True	True	True

fonte: o Autor

```

IDLE Shell 3.12.1
File Edit Shell Debug Options Window Help
>>> A = 0
>>> B = 1
>>> A < 0 and B < 0 # ambas falsas
False
>>> A < 0 and B == 1 # falso e verdadeiro
False
>>> A == 0 and B < 0 # verdadeiro e falso
False
>>> A == 0 and B == 1 # ambas verdadeiras
True
Ln: 13 Col: 0

```

A figura 4.4 define e exemplifica o uso do operador lógico or.

Figura 4.4 – Tabela verdade da operação or

Tabela verdade do operador or Definição: O resultado será verdadeiro se pelo menos uma das condições C1, C2 for verdadeira		
Condição C1	Condição C2	C1 or C2
False	False	False
False	True	True
True	False	True
True	True	True

fonte: o Autor

```

IDLE Shell 3.12.1
File Edit Shell Debug Options Window Help
>>> A = 0
>>> B = 1
>>> A < 0 or B < 0 # ambas falsas
False
>>> A < 0 or B == 1 # falso e verdadeiro
True
>>> A == 0 or B < 0 # verdadeiro e falso
True
>>> A == 0 or B == 1 # ambas verdadeiras
True
Ln: 13 Col: 0

```

Observação:
A linguagem Python não possui o operador lógico **xor** (ou exclusivo) que costuma estar disponível em outras linguagens. Se você precisar usar esse operador deverá usar a expressão equivalente:
$$C1 \text{ xor } C2 = (\text{not } C1 \text{ and } C2 \text{ or } C1 \text{ and not } C2)$$

4.4 CONDIÇÕES COMPOSTAS MISTAS

Esse tipo de condição é o caso mais geral possível e ocorre quando em uma única condição composta você precisa usar **not**, **and** e **or** de forma combinada.

Quando isso ocorre é preciso que você preste atenção à precedência com que esses operadores são considerados. Existe uma ordem de prioridade a ser respeitada. Essa prioridade segue a seguinte ordem: **not** primeiro; **and** em seguida; **or** por último.

É possível usar parênteses para alterar a ordem de prioridade na avaliação dessas expressões. Uma vez inseridos, os parênteses modificam as prioridades e estabelecem qual (ou quais) parte(s) serão avaliada(s) primeiro. Veja no exemplo 4.4 que a colocação ou não dos parênteses afeta o resultado da condição mista.

Exemplo 4.4



```
>>> A = 15
>>> B = 9
>>> C = 9
>>> B == C or A < B and A < C      # caso 1: Resultará Verdadeiro
True
>>> (B == C or A < B) and A < C    # caso 2: Resultará Falso
False
```

(exemplo interativo feito com IDE Idle)

4.5 COMANDO CONDICIONAL – FORMA COMPLETA

Agora que já vimos as condições em detalhes vamos retomar as explicações sobre o comando condicional. Nas linhas abaixo está sua forma completa.

```
if <condição 1>:
    <bloco de comandos 1>
elif <condição 2>:
    <bloco de comandos 2>
elif <condição 3>:
    <bloco de comandos 3>
...
else:
    <bloco de comandos do else>
```

As partes **if** e **else** já foram abordadas na seção 4.1.1 e 4.1.2. A parte **elif** possibilita o uso de múltiplos critérios de decisão ao permitir que condições adicionais sejam usadas.

A execução deste comando segue a seguinte lógica:

- inicia-se pela avaliação da <condição 1> e se ela for verdadeira será executado o <bloco de comandos 1> e pulam-se todos os demais;
- caso a <condição 1> seja falsa, passa-se para a avaliação da <condição 2> e se ela for verdadeira será executado o <bloco de comandos 2>, pulando-se os demais;
- caso a <condição 2> seja falsa, passa-se para a avaliação da <condição 3> e se ela for verdadeira será executado o <bloco de comandos 3>, pulando-se os demais;

- e assim segue-se sucessivamente.
- ao final, se nenhuma das condições postas for verdadeira, então executa-se o <bloco de comandos do else>.

Essa é a forma completa do comando. Porém, as partes `elif` e `else` são opcionais e o programador poderá omiti-las caso não necessite delas. Quando a quantidade de `elif` a serem usados, não há limite. Assim, o programador pode inserir tantos quantos forem necessários para implementar seu programa.

Veja o exemplo 4.5 no qual foi usada a forma completa do comando `if`.

Exemplo 4.5



```
PH = float(input("Digite um valor do pH: "))
if PH < 6.0:
    r = "ácida"
elif PH < 7.0:
    r = "levemente ácida"
elif PH == 7.0:
    r = "neutra"
elif PH < 8.0:
    r = "levemente alcalina"
else:
    r = "alcalina"
print(f'Com pH = {PH} a solução é {r}')
```

Digite um valor do pH: 4.5
Com pH = 4.5 a solução é ácida

Neste exemplo é lido um número real que representa o valor de pH de uma solução. Há 5 possibilidades de classificação conforme o valor do pH conforme o quadro 4.3.

Quadro 4.3 – Faixas de pH para o exemplo 4.5

pH	Classificação da solução
pH menor que 6.0	Solução ácida
pH entre 6.0 (inclusive) e menor que 7.0	Solução levemente ácida
pH igual a 7.0	Solução neutra
pH entre 7.0 e 8.0 (exclusive)	Solução levemente alcalina
pH maior que 8	Solução alcalina

É exatamente isso que está implementado no exemplo 4.5, onde foi usada a construção `if-elif-else` para avaliar o valor fornecido para o objeto `pH` e a partir daí carregar o objeto `r` com a classificação da solução.

4.6 COMANDOS CONDICIONAIS ANINHADOS

É muito comum a necessidade de colocar um segundo comando `if` dentro de outro `if` ou `else`.

Isso está demonstrado no exemplo 4.6, cujo enunciado é: escreva um programa que forneça o tipo de aplicação financeira adequado a um investidor a partir de dois dados fornecidos: o grau de aceitação de risco e o valor a ser investido. O quadro 4.4 mostra as opções de combinação entre esses dois dados. O grau de aceitação de risco deve ser lido do teclado na forma `BX` para baixo ou `AL` para alto. Se for fornecido algo diferente disso o programa deve mostrar uma mensagem indicando que foi fornecido dado inválido. Para o valor deve-se ler um número real.

Quadro 4.4 – Quadro para o exemplo 4.5

Aceitação de risco	Valor < 1000,00	Valor >= 1000,00
Baixo (BX)	Poupança	Renda fixa
Alto (AL)	Bitcoins	Ações

Na solução implementada nas duas linhas iniciais foi feita a leitura dos dados de entrada. Em seguida um primeiro `if` é usado para verificar o que foi digitado para o objeto `risco`; se estiver errado, o programa exibe a mensagem e termina.

Se estiver correto o programa segue para o `else`. Dentro desse `else` do primeiro `if` é escrito todo o resto do código. Um segundo `if` é usado para decidir se o risco é baixo ou alto. Em ambos os casos mais um `if` é usado para estabelecer o tipo de aplicação conforme o que tenha sido digitado para o objeto `valor`.

Exemplo 4.6



```
risco = input('Digite BX ou AL para o grau de risco: ')
valor = float(input('Digite o valor: '))
if risco != 'BX' and risco != 'AL':
    print(f'{risco} é inválido para o grau de risco')
else:
    if risco == 'BX':
        if valor < 1000.0:
            tipo = 'Poupança'
        else:
            tipo = 'Renda fixa'
    else: # risco == 'AL'
        if valor < 1000.0:
            tipo = 'Bitcoins'
        else:
            tipo = 'Ações'
    print(f'Você deve investir em {tipo}')
```

primeira execução

```
Digite BX ou AL para o grau de risco: AL
Digite o valor: 2500
Você deve investir em Ações
```

segunda execução

```
Digite BX ou AL para o grau de risco: AL
Digite o valor: 500
Você deve investir em Bitcoins
```

terceira execução

```
Digite BX ou AL para o grau de risco: BX
Digite o valor: 2500
Você deve investir em Renda Fixa
```

quarta execução

```
Digite BX ou AL para o grau de risco: BX
Digite o valor: 500
Você deve investir em Poupança
```

quinta execução

```
Digite BX ou AL para o grau de risco: PP
Digite o valor: 2500
PP é inválido para o grau de risco
```

Nestes programas que exigem a criação de comandos condicionais aninhados preste bastante atenção para que a relação de subordinação entre os comandos fique correta.

4.7 EXERCÍCIOS RESOLVIDOS COM `if-else`

Exercício Resolvido 4.1



Enunciado: Escreva um programa que leia um número inteiro e mostre na tela se ele é par ou ímpar. Lembrando que para saber a paridade de um inteiro é preciso calcular o resto da sua divisão por 2. Se o resto for 0 o número é par, se o resto for 1 o número é ímpar.

```
X = int(input('Digite um inteiro: '))
Resto = X % 2          # Calcula o resto da divisão de X por 2
if Resto == 0:         # Verifica se o resto é 0
    print(f'O número {X} é par')
else:                  # Se a condição resultou False desvia para este else
    print(f'O número {X} é ímpar')
print('Fim do Programa')
```

primeira execução

```
Digite um inteiro: 82
O número 82 é par
Fim do Programa
```

segunda execução

```
Digite um inteiro: 53
O número 53 é ímpar
Fim do Programa
```

Nesta solução, após a leitura de `X`, é feito o cálculo do `Resto` com o uso do operador matemático `%`. Em seguida este resultado é usado para construir a condição do comando condicional `Resto == 0`. Caso a avaliação da condição resulte em verdadeiro apenas o primeiro `print()` será executado, mostrando na tela que o número é par; se resultar em falso será executado apenas o segundo `print()` mostrando que o número é ímpar.

Do código acima destacamos duas linhas, transcritas abaixo:

```
Resto = X % 2
if Resto == 0:
    ...
```

Elas foram escritas desta forma por razões didáticas. Na prática elas poderiam ser condensadas em uma única linha da seguinte forma com a eliminação do objeto `Resto` e sem prejudicar o funcionamento do programa.

```
if X % 2 == 0:
    ...
```

Exercício Resolvido 4.1 - reescrito

```
X = int(input('Digite um inteiro: '))
if X % 2 == 0:          # Calcula o resto e já verifica se é 0
    print(f'O número {X} é par')
else:                  # Se a condição resultou False desvia para este else
    print(f'O número {X} é ímpar')
print('Fim do Programa')
```

Experimente escrever e testar esse código para verificar que funciona como o anterior

Esta segunda forma de fazer é mais usual entre os programadores mais experientes.

Exercício Resolvido 4.2



Enunciado: Escreva um programa que leia dois inteiros e mostre na tela apenas o menor dos dois. Se ambos forem iguais, mostre qualquer um deles.

```
A = int(input('Digite A: '))
B = int(input('Digite B: '))
if A <= B:
    print(f'O menor número é {A}')
else:
    print(f'O menor número é {B}')
print('Fim do Programa')
```

primeira execução

```
Digite A: 23
Digite B: 78
O menor número é 23
Fim do Programa
```

segunda execução

```
Digite A: 58
Digite B: 12
O menor número é 12
Fim do Programa
```

Para exibir o menor valor dentre A e B foi criada a condição `A <= B`. Se ela resultar em verdadeiro significa uma de duas possibilidades: ou A é menor que B; ou A e B são iguais. Neste caso A é exibido – e isso está de acordo com o enunciado que diz "se ambos forem iguais, mostre qualquer um deles". Caso a condição resulte em falso, então B será exibido, pois é menor que A.

Exercício Resolvido 4.3



Enunciado: Altere o programa anterior de modo que ele continue exibindo o menor dos dois valores lidos. Porém, quando forem iguais o programa deve exibir o valor junto com o texto "Os dois números são iguais".

```
A = int(input('Digite A: '))
B = int(input('Digite B: '))
if A == B:
    print(f'Os dois números são iguais e valem {A}')
else:
    if A < B:
        print(f'O menor número é {A}')
    else:
        print(f'O menor número é {B}')
print('Fim do Programa')
```

primeira execução

```
Digite A: 6
Digite B: 6
Os dois números são iguais e valem 6
Fim do Programa
```

segunda execução - teste você mesmo digite um valor menor para A

terceira execução - teste você mesmo digite um valor menor para B

Neste caso existem três situações possíveis: `A == B` ou `A < B` ou `B < A`

Quando isso ocorre são necessários dois comandos condicionais aninhados. Optamos por perguntar primeiro se `A == B` e em caso verdadeiro exibir a mensagem de que são iguais. No caso de serem diferentes a execução é desviada para o `else` do primeiro `if`. Dentro desse `else` restam as outras duas possibilidades:

A menor ou B menor e para decidir sobre elas um segundo `if` foi usado. Escreva esse programa e teste-o com valores que produzam as três situações.

Esse mesmo enunciado pode ser resolvido de outra maneira se for usada parte `elif` do comando condicional. Veja o código 4.3 reescrito a seguir e note que a solução fica bem interessante, pois é mais compacta e produz o mesmo resultado.

Exercício Resolvido 4.3 - reescrito

```
A = int(input('Digite A: '))
B = int(input('Digite B: '))
if A == B:                                     # A e B são iguais
    print(f'Os dois números são iguais e valem {A}')
elif A < B:                                    # A é menor
    print(f'O menor número é {A}')
else:                                          # B é menor
    print(f'O menor número é {B}')
print('Fim do Programa')
```

Experimente escrever e testar esse código para verificar que funciona como o anterior

Exercício Resolvido 4.4



Enunciado: Escreva um programa para exibir na tela o nome e a categoria de um lutador. O programa deve ler um string para o nome e um número real para o peso. Conforme o peso ocorrerá o enquadramento na categoria, segundo esta tabela (fictícia):

Peso (kg)	Categoria	Peso (kg)	Categoria
menor que 52	Inválido	maior ou igual a 79 e menor que 86	Meio-médio
maior ou igual a 52 e menor que 65	Pena	maior ou igual a 86 e menor que 90	Médio
maior ou igual a 65 e menor que 72	Leve	maior ou igual a 90 e menor que 100	Meio-pesado
maior ou igual a 72 e menor que 79	Ligeiro	maior ou igual a 100	Pesado

```
Nome = input('Digite o nome: ')
Peso = float(input('Digite o peso: '))
if Peso < 52:
    Categoria = ''
elif Peso < 65:
    Categoria = 'Pena'
elif Peso < 72:
    Categoria = 'Leve'
elif Peso < 79:
    Categoria = 'Ligeiro'
elif Peso < 86:
    Categoria = 'Meio-médio'
elif Peso < 93:
    Categoria = 'Médio'
elif Peso < 100:
    Categoria = 'Meio-pesado'
else:
    Categoria = 'Pesado'
msg = 'O lutador {} pesa {:.3f} kg e se enquadra na categoria {}'
if Categoria != '':
    print(msg.format(Nome, Peso, Categoria))
else:
    print(f'Peso inválido: {Peso}')
print("Fim do programa")
```

primeira execução

```
Digite A: 6
Digite B: 6
Os dois números são iguais e valem 6
Fim do Programa
```



```
segunda execução
Digite o nome: Felipe
Digite o peso: 43.2
Peso inválido: 43.2
Fim do programa

terceira execução
Digite o nome: Maguila
Digite o peso: 102.3
O lutador Maguila pesa 102.300 kg e se enquadra na categoria Pesado
Fim do programa
```

Este exercício resolvido 4.4 tem o propósito de ilustrar uma situação também frequente em que há várias alternativas de seleção. Na implementação dessa solução optou-se por usar a forma completa do comando condicional, que inclui a parte `elif`. Também seria possível resolver este problema usando `ifs` aninhados. Funcionaria perfeitamente, mas não seria muito prático pois criaria 8 níveis de indentação e com uma legibilidade não muito boa.

Verifique que existe o caso em que o peso é inválido quando < 52 e os demais casos para pesos maiores. Na solução optou-se por carregar o objeto `Categoria` com um string vazio e depois no final do programa essa informação foi usada em um `if` para exibir na tela a mensagem final.

Você deve ter percebido que temos dado preferência para usar `f-strings` na formatação das mensagens de saída. Porém neste caso, usamos o método `.format()` também foi usado. Isso foi feito para mostrar que os dois modos podem coexistir em um programa.

Exercício Proposto 4.1

Enunciado: *Classificação indicativa é um conceito que se aplica à faixa etária para a qual uma obra audiovisual se recomenda ou não. Suponha que um filme em cartaz no cinema tenha a Classificação de 16 anos. Escreva um programa que leia a idade de uma pessoa e mostre se está de acordo ou não com a classificação.*

Exercício Proposto 4.2

Enunciado: *Escreva um programa que leia um número inteiro e mostre na tela se ele é divisível por 10 ou não.*

Exercício Proposto 4.3

Enunciado: *Uma empresa financeira concede empréstimos a pessoas físicas quando o valor da parcela é menor que 8% do salário da pessoa. Escreva um programa que leia dois números reais: o valor do salário e o valor da parcela e informe se o empréstimo será concedido ou não.*

Exercício Proposto 4.4

Enunciado: *Escreva um programa que leia o nome de um aluno e as notas obtidas em três avaliações. A média final é a média aritmética das três notas e a pessoa estará aprovada se essa média for maior ou igual a 7.0. Mostre na tela o nome, a média e a situação que será "Aprovado" ou "Reprovado".*

Exercício Proposto 4.5

Enunciado: Escreva um programa que leia a idade de uma pessoa e indique qual sua classe eleitoral:

- a) menor que 16 anos -> não eleitor
 - b) entre 18 completos e 65 anos incompletos -> eleitor obrigatório
 - c) entre 16 anos completos e 18 anos incompletos ou 65 anos completos -> eleitor facultativo
-

Exercício Proposto 4.6

Enunciado: Escreva um programa para uma fábrica de calçados que leia o código LL de um calçado, que é um número inteiro com 2 dígitos. Exiba na tela a linha do calçado, conforme a tabela a seguir. Se o número fornecido não estiver na tabela, deve-se exibir a mensagem "Código inválido".

LL	Linha de calçados	LL	Linha de calçados
16	Bebê	49	Masculino esportivo
23	Infantil feminino	52	Feminino formal salto baixo
25	Infantil masculino	53	Feminino formal salto alto
29	Infantil esportivo	55	Feminino casual salto baixo
42	Masculino formal	56	Feminino casual salto alto
43	Masculino casual	59	Feminino esportivo

Exercício Proposto 4.7

Enunciado: Em Albalândia mulheres e homens podem servir o exército do país. O serviço é opcional e é muito comum que as pessoas se apresentem para o serviço em algum momento da vida. Existe uma única restrição para ingresso que é a idade da pessoa: para mulheres a idade aceita é entre 21 e 34 anos; para homens a idade aceita é entre 18 e 39 anos. Escreva um programa que leia três dados de entrada: nome da pessoa, idade e sexo e informe se a pessoa será aceita ou não para o serviço.

Para o sexo deve ser lido apenas 1 caractere que pode ser 'f' ou 'F' para feminino e 'm' ou 'M' para masculino, qualquer coisa diferente deve ser informado como inválido.

Exercício Proposto 4.8

Enunciado: Escreva um programa que leia um número inteiro que representa um ano. Informe se esse ano é bissexto ou não.

Regra: O ano é bissexto se cumprir uma das seguintes condições:

- a) ser múltiplo de 4 e ao mesmo tempo não ser múltiplo de 100
 - b) ser múltiplo de 400
-

Exercício Proposto 4.9

Enunciado: Escreva um programa que leia 3 números inteiros e mostre na tela uma das seguintes opções:

- a) "Os três valores são iguais"
 - b) "Há dois valores iguais e um diferente"
 - c) "Os três valores são diferentes"
-

Exercício Proposto 4.10

- Enunciado:** *Escreva um programa que leia 3 números inteiros e mostre na tela se eles formam um triângulo ou não. Caso formem um triângulo informe o tipo de triângulo (equilátero, isósceles ou escaleno).*
- Regra:** *Para três números formarem um triângulo precisa ocorrer que:*
- a) os três números precisam ser maiores que zero;*
 - b) a soma dos dois menores valores deve ser maior que o terceiro.*
-

Exercício Proposto 4.11

- Enunciado:** *No comércio, o conceito de Margem Bruta é uma porcentagem que é aplicada ao preço de custo para se obter o preço de venda. Uma loja tem como política comercial aplicar uma margem bruta de 45% quando o preço de custo de um produto é menor ou igual a R\$ 100,00. Se o produto custa mais que isso a margem bruta é de 35%. Escreva um programa que leia o preço de custo do produto e mostre na tela qual o seu preço de venda, com duas casas decimais.*
-

Exercício Proposto 4.12

- Enunciado:** *Leia um número inteiro entre 1 e 12 e exiba o mês correspondente. Caso seja digitado um número fora desse intervalo, o programa deve exibir uma mensagem informando que não existe mês com este número.*
-

Exercício Proposto 4.13

- Enunciado:** *Nas eleições municipais os municípios com 200 mil eleitores ou mais tem segundo turno caso o primeiro colocado não tenha mais do que 50% dos votos. Escreva um programa que leia o nome do município, a quantidade de eleitores e a quantidade de votos do candidato mais votado e informe se haverá segundo turno ou não.*
-

Exercício Proposto 4.14

- Enunciado:** *Em um determinado momento do dia a cotação de compra das moedas estrangeiras é a seguinte:*
- Dólar:** US\$ 1.00 = R\$ 4.89 - **Euro:** € 1.00 = R\$ 5.26 - **Libra Esterlina:** £ 1.00 = R\$ 6.17
- Escreva um programa que leia o tipo (D, E ou L maiúsculo) e o valor de moeda estrangeira que se quer comprar e calcule o valor em reais necessários.*
-

Capítulo 5

COMANDOS DE REPETIÇÃO

Muitas vezes um determinado bloco de código precisa ser repetido várias vezes. A esta situação de execução de repetições em um programa damos o nome de "laço de repetição" ou "loop de repetição". Na linguagem Python existem dois comandos que realizam repetições:

- Comando `while`: este é o comando de repetição de uso geral e será visto neste capítulo;
- Comando `for`: este comando é para uso especializado e será visto no capítulo 7;

5.1 O COMANDO `while`

5.1.1 COMO IMPLEMENTAR UM LAÇO DE REPETIÇÃO EM PYTHON

O comando `while` em Python tem a construção básica abaixo que pode ser interpretada como: "enquanto a condição for verdadeira execute o conjunto de comandos".

```
while <condição>:  
    <conjunto de comandos>
```

A condição segue exatamente as mesmas regras utilizadas nas condições já vistas no capítulo 4 quando tratamos do comando `if-else`. Os comandos subordinados ao `while`, podem ser quaisquer comandos válidos em Python, em qualquer quantidade e extensão. Assim como no comando `if-else`, a indentação é importante pois define a relação de subordinação entre o comando e seu bloco de código subordinado.

Para exemplificar a implementação de um laço considere o exemplo 5.1 a seguir no qual é feita a exibição de todos os números inteiros entre 1 e 10, sendo um valor em cada linha.

Neste exemplo, na linha 1 é definido um objeto identificado por `cont` e inicializado com o valor 1. Na linha 2 está o comando `while` construído com a condição `cont <= 10`. A avaliação dessa condição resulta em `True` (verdadeiro) de modo que o conjunto de comandos subordinado, constituído pelas linhas 3 e 4, é executado uma primeira vez. Com isso, o valor inicial de `cont` é exibido (linha 3) e 1 é somado a `cont` (linha 4), que passará a ser 2.

Em seguida o programa retorna para a linha 2, a condição é avaliada e novamente resultará `True`, pois `cont` é menor que 10. Isto fará com que o `print` e a soma de 1 em `cont` sejam executados uma segunda vez.

Com isso `cont` passará a conter o valor 3 e o laço prosseguirá com `cont` avançando de um e um até o final. Quando `cont` atingir o valor 11 a condição `cont <= 10` se tornará falsa e o `while` terminará. Ao final desse processo dez linhas terão sido exibidas na tela contendo os valores de 1 a 10.

Exemplo 5.1



```
print("Início do Programa")
cont = 1                      # linha 1
while cont <= 10:             # linha 2
    print(cont)               # linha 3
    cont = cont + 1           # linha 4
print("Fim do Programa")
```

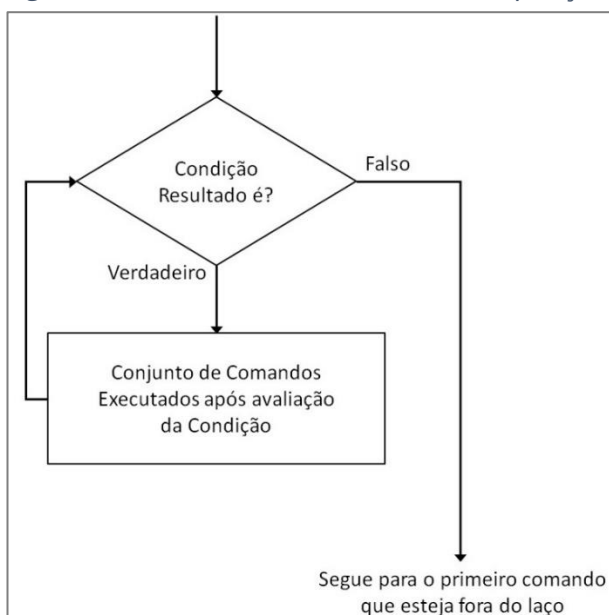
```
Início do Programa
1
2
3
4
5
6
7
8
9
10
Fim do Programa
```

5.1.2 FLUXO DE EXECUÇÃO DE LAÇOS DE REPETIÇÃO `while`

No exemplo 5.1 foi mostrado como usar o comando `while` e agora vamos aprofundar nos aspectos conceituais dele.

O primeiro ponto é saber que o teste da condição é feito no início do laço. A figura 5.1 ilustra esta situação: a avaliação da condição é feita antes de se executar o conjunto de comandos subordinado. Este fato tem uma implicação conceitual importante porque nos casos em que a condição for previamente falsa o conjunto subordinado não será executado nenhuma vez.

Figura 5.1 – Conceito do Comando de Repetição



fonte: o Autor

Todo laço para ser implementado requer quatro elementos:

- Inicialização: situação inicial do controle do laço;
- Condição de continuidade do laço;
- Iteração: ação sobre o controle do laço, a cada repetição; e
- Corpo: bloco de código subordinado.

Os três primeiros dizem respeito à estrutura e controle do laço. A inicialização constitui-se de todo código necessário para determinar a situação inicial do laço. A condição de continuidade é uma expressão lógica, simples ou composta, cujo resultado é avaliado em falso ou verdadeiro a cada repetição e que determinará se o laço termina ou prossegue, respectivamente. A iteração é todo comando (um ou mais de um) que modifica os objetos envolvidos na condição de continuidade, a cada execução do laço.

Por fim, o bloco de código subordinado é constituído pelos comandos que devem ser executados repetidas vezes. No exemplo 5.1 acima, a inicialização está na linha 1, a condição de continuidade está na linha 2 e a iteração é a linha 4. O corpo do laço é a linha 3.

Agora vamos resolver alguns exercícios para fixar esses conceitos.

5.2 EXERCÍCIOS RESOLVIDOS COM WHILE

Exercício Resolvido 5.1



Enunciado: Escreva um programa que permaneça em laço enquanto um valor *X* lido for diferente de zero. Para cada valor de *X* presente na tela se é par ou ímpar.

```
X = 1                                # linha 1
while X != 0:                        # linha 2
    X = int(input('Digite X: '))     # linha 3
    if X % 2 == 0:                   # linha 4
        print(f'{X} é par')          # linha 5
    else:                             # linha 6
        print(f'{X} é ímpar')        # linha 7
print("Fim do Programa")
```

Digite X: 34
34 é par
Digite X: 21
21 é ímpar
Digite X: 0
0 é par
Fim do Programa

Nesta solução o controle do laço é realizado de forma diferente. Desta vez não temos um contador, ou seja, a natureza do controle do laço é de outro tipo.

Neste caso não sabemos quantas vezes o laço irá repetir, pois depende dos dados que o usuário for digitando. O que se sabe apenas é que o laço deve terminar quando zero for digitado para *X*.

Para garantir que o laço seja iniciado, o objeto *X* deve ser criado contendo qualquer valor diferente de 0, o que é feito na linha 1. A linha 1 é a linha de inicialização. A iteração é implementada na linha 3 que altera o valor do objeto *X*. O novo *X* lido logo no início do laço também é usado no corpo do mesmo, que é constituído pelas linhas 4 a 7. O resto da divisão de *X* por 2 é calculado e comparado com zero. Se o resultado dessa

comparação for verdadeiro, então o número é par, caso contrário é ímpar. Quando zero for digitado, o programa dirá que zero é par e terminará.

Exercício Resolvido 5.2



Enunciado: Escreva um programa que mostre na tela a tabuada do número inteiro N que deve ser lido do teclado.

```
N = int(input('Digite N: '))
cont = 1
while cont <= 10:
    R = cont * N
    print(f'{cont} x {N} = {R}')
    cont = cont + 1
print('Fim do Programa')
Digite N: 3
1 x 3 = 3
2 x 3 = 6
3 x 3 = 9
4 x 3 = 12
5 x 3 = 15
6 x 3 = 18
7 x 3 = 21
8 x 3 = 24
9 x 3 = 27
10 x 3 = 30
Fim do Programa
```

Neste exercício temos novamente o controle do laço feito com o uso de um contador. O objeto de controle do laço é o `cont`, inicializado com 1 e que é incrementado a cada repetição até que assuma um valor maior que 10. Para cada repetição é feito o cálculo do resultado da linha da tabuada e sua exibição.

Exercício Resolvido 5.3



Enunciado: Escreva um programa que mostre na tela os 10 primeiros termos de uma progressão aritmética (PA) com primeiro termo P e razão R . Os dois números P e R são inteiros e devem ser lidos do teclado.

```
P = int(input("Digite o primeiro termo: ")) # linha 1
R = int(input("Digite a razão: "))          # linha 2
cont = 0                                    # linha 3
while cont < 10:                             # linha 4
    print(P)                                  # linha 5
    P = P + R                                # linha 6
    cont = cont + 1                           # linha 7
print('Fim do Programa')
Digite o primeiro termo: 4
Digite a razão: 5
4
9
14
19
24
29
34
39
44
49
Fim do Programa
```

Nas duas primeiras linhas é feita a leitura dos dados. Em seguida é implementado o laço de repetição com o objeto `cont` inicializado em 0; a condição de continuidade `cont < 10`; e o incremento

`cont = cont + 1` alterando o objeto de controle do laço. A cada repetição um termo da PA é exibido e o próximo é calculado.

Propositadamente, neste exercício o objeto `cont` foi inicializado com 0 e a condição foi escrita como `cont < 10`. Compare-a com o exemplo 5.1 onde o objeto `cont` foi iniciado com 1 e a condição escrita como `cont <= 10`. Estas são duas formas diferentes de implementar um laço que executa o mesmo número de vezes. Em situações como essa cabe ao programador escolher a alternativa que considera mais interessante para o programa.

Exercício Resolvido 5.4



Enunciado: Escreva um programa que leia do teclado um número inteiro *D*. Esse número deve ser obrigatoriamente maior que zero. Em seguida exiba na tela todos os números inteiros menores que 100 e que sejam divisíveis por *D*.

```
D = int(input('Digite D: '))
if D <= 0:
    print(f'O valor {D} é inválido')
else:
    i = 1
    while i < 100:
        if i % D == 0:
            print(i)
        i = i + 1
print('Fim do Programa')
```

```
Digite D: 22
22
44
66
88
Fim do Programa
```

Neste exercício temos uma novidade: o valor de *D* deveria ser maior que zero. Com isso usamos um comando `if` para fazer a verificação e se *D* for menor ou igual a zero emitimos uma mensagem dizendo que é inválido. No `else` (que ocorrerá quando *D* for maior que zero) foi implementado o restante do código. Neste ponto foi implementado um laço de repetição baseado em um contador. O objeto `i` foi usado no controle do laço e cada vez que esse `i` era divisível por *D* seu valor foi exibido na tela. No teste exibimos todos os números inteiros até 100 e divisíveis por 22.

Este exercício também tem o propósito de mostrar que podem existir comandos condicionais dentro de comandos de laço e vice-versa, sem limite de vezes em que isso ocorre.

Exercício Resolvido 5.5



Enunciado: Escreva um programa que permaneça em laço enquanto um valor inteiro lido for diferente de zero. Totalize e conte os valores digitados, exceto o zero, e apresente esses valores na tela. Totalizar é somar os valores.

```
soma = qtde = 0
A = 1
while A != 0:
    A = int(input("Digite X: "))
    if A != 0:
        soma = soma + A
        qtde = qtde + 1
print(f'Soma dos valores = {soma}')
```

```
print(f'Quantidade = {qtde}')
print('Fim do Programa')
Digite X: 16
Digite X: 40
Digite X: 21
Digite X: 6
Digite X: 0
Soma dos valores = 83
Quantidade = 4
Fim do Programa
```

Neste programa o controle do laço se faz através do objeto `A` que é lido a cada repetição. Quando for digitado o valor 0 para `A` o laço irá terminar e é preciso o cuidado de não somar 1 no objeto `qtde` quando isso ocorrer. Por esse motivo foi usado o comando condicional `if A != 0` dentro do laço.

5.3 MAIS DETALHES SOBRE LAÇOS EM PYTHON

5.3.1 COMANDO CONTINUE

O comando `continue` altera o fluxo normal de execução de um laço de repetição. Ele é usado para interromper uma repetição que esteja em curso dentro de um laço e avançar para a próxima repetição. O exemplo 5.2 foi criado para exibir valores de 1 a 5 na tela, com exceção do 4. O objeto `i` é usado no controle do laço e também é exibido na tela. Porém, quando ele assume o valor 4, o `if` dentro do laço resulta verdadeiro e o comando `continue` é executado interrompendo a atual execução e seguindo para a próxima. Como o `print()` está após esse código ele acaba sendo pulado e o valor 4 não é exibido na tela.

Exemplo 5.2



```
i = 0
while i < 5:
    i = i + 1
    if i == 4:
        continue
    print(i)
```

Quando o `continue` é executado a execução é imediatamente retornada ao cabeçalho do comando de repetição

1
3
4
5

É preciso ter muito cuidado com esse comando, pois é muito comum que o programador inexperiente cometa erros sérios ao usá-lo de forma inadequada. Como exemplo veja as linhas a seguir:

```
i = 0
while i < 5:
    if i == 4:
        continue
    i = i + 1
    print(i)
```

A mudança de posição desta linha é a única alteração.

Uma simples alteração de posição na linha `i = i + 1` e este código fica totalmente errado. Quando o valor de `i` chegar a 4 será executado o `continue` **sem passar pela linha do incremento de `i`**. Isso implica que o valor de `i` será 4 para sempre e esse laço executará indefinidamente. É o que se chama de laço infinito, um erro severo, pois seu programa parecerá congelado, não exibindo nada na tela e nem respondendo ao usuário. Faça o teste. Escreva esse código e rode. Ele mostrará na tela os valores de 1 a 4 e depois "congelará".

5.3.2 COMANDO BREAK

O comando `break` também altera o fluxo de execução de um laço de repetição. Quando executado o `break` o laço é encerrado imediatamente. No exemplo 5.3 fizemos um laço infinito por definição ao escrever `while True`. No entanto, observe que dentro do laço está sendo usado o comando `break`, o que significa que esse laço não é verdadeiramente infinito. Quando o usuário digitar 0 (zero) o `if` dentro do laço resultará verdadeiro e o `break` será executado interrompendo o laço.

Exemplo 5.3



```
X = 1
while True:
    X = int(input('Digite X: '))
    if X == 0:
        print('  você digitou zero...')
        break
    print(X)
print('Fim do Programa')
```

Quando o `break` é executado o laço termina imediatamente

Digite X: 8
8
Digite X: -4
-4
Digite X: 1
1
Digite X: 0
você digitou zero...
Fim do Programa

5.3.3 CLÁUSULA `else` DO COMANDO `while`

Em Python a forma completa do comando `while` inclui uma parte `else`, que parece muito estranha para os programadores que aprenderam a programar com outras linguagens e só depois conhecem o Python. Então, a forma geral do `while` é:

```
while <condição>
    <bloco de comandos 1 - que pode conter um break>
else:
    <bloco de comandos 2>
```

Seu funcionamento ocorre assim: o laço é repetido normalmente enquanto a condição for verdadeira. Quando a condição se tornar falsa o laço termina e o código do `else` é executado. Se um comando `break` existir no laço e for executado, então o `else` não é executado.

Exemplo 5.4

```
X = 1
while X > 0:      # enquanto X for positivo faça as repetições
    X = int(input('Digite X: '))
    if X == 0:    # se X for zero interrompa o laço
        print('  você digitou zero...')
        break
    print(X)
else:             # este else é executado quando X for negativo se X for zero não
    print('você digitou negativo')
print('Fim do Programa')
```

Escreva este programa e teste-o várias vezes, algumas digitando 0 e outras um negativo

O exemplo 5.4 é uma adaptação do 5.3, no qual foram feitas duas alterações: a troca da condição do `while` e a inclusão da cláusula `else` para exibir a mensagem "você digitou negativo". Teste esse código para ver seu funcionamento.

Exercício Proposto 5.1

Enunciado: *Reescreva o Exercício Resolvido 5.5 de modo a eliminar o comando `if` que foi acrescentado dentro do laço `while`. Procure pensar em uma forma de eliminar esse condicional e ao mesmo tempo manter o programa correto, totalizando e contando os valores diferentes de zero que forem digitados.*

Dica: a solução consiste em alterar a ordem dos comandos existentes dentro do laço `while`.

Exercício Proposto 5.2

Enunciado: *Escreva um programa que leia um número N e em seguida exiba na tela todos os números divisíveis por 7 entre 1 e N (inclusive).*

Exercício Proposto 5.3

Enunciado: *Escreva um programa que obrigatoriamente leia um inteiro que esteja no intervalo fechado $[100, 200]$. Se o valor fornecido estiver fora do intervalo o programa deve avisar que o valor é inválido e permanecer no laço. Quando um valor válido for fornecido o programa deve informar que o valor foi aceito e terminar.*

Exercício Proposto 5.4

Enunciado: *Escreva um programa que leia dois números inteiros: L_{Min} e L_{Max} . Em seguida exiba na tela todos os valores dentro do intervalo fechado $[L_{Min}, L_{Max}]$.*

Exercício Proposto 5.5

Enunciado: *Escreva um programa que leia três números inteiros: L_{Min} , L_{Max} e D . Em seguida exiba na tela todos os valores divisíveis por D que estão dentro do intervalo fechado $[L_{Min}, L_{Max}]$.*

Exercício Proposto 5.6

Enunciado: *Escreva um programa que permaneça em laço lendo cadeias de caracteres (strings). Para cada cadeia digitada o programa deve exibir a cadeia seguida da quantidade de caracteres que ela contém. O programa termina quando for digitado "FIM" (em letras maiúsculas).*

Exercício Proposto 5.7

Enunciado: *Escreva um programa que permaneça em laço lendo quantidades (números inteiros) de produtos vendidos. O laço termina quando for digitado zero ou um valor negativo. Ao término do laço exiba na tela a soma de todas as quantidades digitadas (se for digitado um negativo para sair do laço ele não deve afetar o total).*

Exercício Proposto 5.8

Enunciado: Uma indústria metalúrgica adota um código de produto com o seguinte formato TMMM, onde T indica o uso do produto, sendo 1 para residencial; 2 para industrial e MMM indica qual é o produto.

Escreva um programa que permaneça em laço até que seja digitado 0. Em cada repetição leia duas informações:

- a) O código do produto;
- b) A quantidade vendida desse produto

O programa deve totalizar separadamente e exibir na tela as quantidades de produtos residenciais e industriais vendidos. Se o dígito T do código não for 1 ou 2 deve ser mostrado "Tipo Inválido" e a quantidade deve ser ignorada.

Exercício Proposto 5.9

Enunciado: Escreva um programa que leia um número inteiro N. Em seguida calcule e mostre na tela o fatorial de N (N!).

Exercício Proposto 5.10

Enunciado: Escreva um programa que leia um número inteiro e informe se esse número é primo ou não. Lembrando: um número primo é divisível apenas por 1 e por ele mesmo.

Exercício Proposto 5.11

Enunciado: Escreva um programa que leia uma quantidade Qtde e mostre na tela os Qtde primeiros termos da sequência de Fibonacci.

A sequência de Fibonacci é definida da seguinte forma: a) os dois primeiros termos da sequência são 0 e 1. Do terceiro termo em diante cada termo é a soma dos dois anteriores.

Caso de teste: Se Qtde = 9, então a sequência é: 0, 1, 1, 2, 3, 5, 8, 13, 21

Exercício Proposto 5.12

Enunciado: Escreva um programa que leia dois inteiros: Qtde e Prim. Em seguida mostre na tela os Qtde termos da sequência de Fibonacci que sejam maiores que Prim.

Capítulo 6

TRATAMENTO DE EXCEÇÕES

6.1 CONCEITO

Erros podem acontecer em um sistema computacional. Há erros de hardware, outros relacionam-se ao sistema operacional e há os erros dos programas de usuário final. Este último é o que nos interessa.

O tratamento de exceções está relacionado com erros que podem acontecer em um programa. Esse termo, **exceção**, se refere às situações que necessitam de atenção e algum tratamento especial fora da lógica normal de processamento do programa. Essas situações estão sempre relacionadas aos dados que estão nos objetos e são decorrentes uma grande variedade de fatores, tais como:

- Divisão por zero;
- Erro de dados de entrada;
- Indexadores de array fora dos limites corretos;
- Erros da aplicação permitindo valores e formatos incorretos nos objetos; entre outros.

A pequena lista acima não esgota todas as possibilidades de erros e falhas que podem ocorrer com os dados em um programa de computador. E quando qualquer desses fatores ocorre, o programa pode parar de funcionar podendo ocasionar vários tipos de contratempo e prejuízo. A elaboração de uma boa estratégia de tratamento de exceções permite a criação de programas robustos e confiáveis.

No capítulo 4 o exemplo 4.1 contém um erro que interrompeu sua execução. É este código:

Exemplo 4.1 – rerepresentação do exemplo do capítulo 4

```
A = 10
B = 0
R = A / B
print(R)

Traceback (most recent call last):
  File "D:\exemplos\exemplo_4.1.py", line 3, in <module>
    R = A / B
    ~~~~
ZeroDivisionError: division by zero
```

Neste exemplo ocorre um erro da classe `ZeroDivisionError` decorrente da tentativa de fazer uma divisão por zero. Os exemplos 6.1 e 6.2 a seguir mostra mais dois erros de natureza distinta.

No exemplo 6.1 o objeto `Dado` foi carregado com o string `'13o5'` e em seguida houve a tentativa de convertê-lo para número inteiro. É uma situação que pode acontecer em que, por um engano de digitação do usuário, o programa recebeu uma letra `'o'` no lugar do algarismo `'0'` e em virtude disso não é possível fazer a conversão. O erro gerado foi da classe `ValueError` e o programa parou de funcionar.

Exemplo 6.1

[Teste este código no Idle](#)

```
Dado = '13o5'
Valor = int(Dado)
print(Valor)

Traceback (most recent call last):
  File "exemplo_6.1.py", line 2, in <module>
    Valor = int(Dado)
           ^^^^^^^^^
ValueError: invalid literal for int() with base 10: '13o5'
```

No exemplo 6.2 existe uma lista que contém 5 elementos e houve a tentativa de exibir na tela o elemento de índice 10, que não existe. Como resultado ocorreu um erro da classe `IndexError`.

Exemplo 6.2

[Teste este código no Idle](#)

```
L = [12, 34, 56, 67, 89]
print(L[10])

Traceback (most recent call last):
  File "exemplo_6.2.py", line 8, in <module>
    print(L[10])
          ~^^^^
IndexError: list index out of range
```

Você pode estar pensando que está evidente que são erros simples e que basta um pouco de atenção ao programador para não os cometer. Ocorre, no entanto, que as falhas nos programas do mundo real não são evidentes dessa forma. O caso 6.1, por exemplo, poderia ser um software rodando na internet e que recebe esse dado com a letra `'o'` digitado por um usuário com pouca familiaridade com computadores. No caso 6.2 o índice 10 não estaria ali dessa forma explícita, mas sim dentro de um objeto que foi carregado a partir de um banco de dados, no qual podem existir inconsistências de cadastro. Em outras palavras, existem inúmeras situações que podem ser exemplificadas e que vão mostrar que falhas ocorrem.

Considere novamente o exemplo 4.1. No capítulo 4 resolvemos o erro de divisão por zero com o uso de um comando condicional `if`. Então você poderia questionar: poderíamos sempre evitar os erros usando `ifs`? Não é tão simples assim.

Os sistemas computacionais estão cada vez mais complexos e usar condicionais para prever e tratar tudo o que pode dar errado só vai fazer com que esse sistema fique cheio de condicionais e ainda mais complexo, pois tudo o que pode dar errado teria que ser previsto e tratado. Assim, a maioria das linguagens de programação possui recursos voltados para a identificação e o tratamento desses erros, cabendo ao profissional de programação dominar as técnicas e métodos relacionados a isso.

6.2 TRATAMENTO DE EXCEÇÕES EM PYTHON – FORMA ESSENCIAL

Para implementar o tratamento de exceções em Python utiliza-se o comando `try-except`. As linhas a seguir mostram a estrutura deste comando:

```
try:
    <bloco de código protegido>
except:
    <bloco de tratamento da exceção>
```

O conceito envolvido neste comando é a proteção de um determinado bloco de código. A palavra-chave `try` dá início ao código protegido. Se algum erro ocorrer em qualquer linha deste bloco a execução é desviada para o bloco sob a cláusula `except`.

Vamos ver essa estrutura na prática através do exemplo 6.3.

Exemplo 6.3



```
A = int(input('Digite A: '))
B = int(input('Digite B: '))
try:
    R = A / B
    print(f'Resultado = {R}')
except:
    print('Não é possível calcular a divisão')
```

primeira execução

```
Digite A: 20
Digite B: 0
Não é possível calcular a divisão
```

segunda execução

```
Digite A: 75
Digite B: 4
Resultado = 18.75
```

Nesse código as leituras de A e B estão fora do `try` e não estão protegidas. O cálculo da divisão e o `print()` estão dentro do `try` e estão protegidos. O `except` define qual será o tratamento em caso de erro na parte protegida.

Na primeira execução deste exemplo foi fornecido o valor 0 para B. Como isso gera o erro de divisão por zero a execução foi desviada para o `except`, dentro do qual é exibida a mensagem "*Não é possível calcular a divisão*". Na segunda execução o valor de B é diferente de zero, então as duas linhas do código protegido pelo `try` foram executadas e o resultado da divisão foi apresentado na tela.

6.2.1 EXCEÇÕES NOMEADAS

Toda exceção em Python é uma classe e tem um identificador. A título de exemplo, no início deste capítulo mostramos três classes de exceções: `ZeroDivisionError`, `ValueError`, `IndexError`

Agora volte ao exemplo 6.3 e olhe para a primeira linha onde é feita a leitura de A. Considere que o usuário quisesse digitar 50, mas colocou a letra 'o' no lugar do zero. Isso vai gerar um `ValueError` e essa linha não está protegida.

Vamos então fazer uma alteração e passar os comandos de leitura para dentro do bloco protegido pelo `try`. E assim temos o exemplo 6.4 no qual o bloco protegido contém quatro linhas de código conforme pode ser visto.

A questão da proteção está resolvida, mas agora temos outro problema. Qualquer erro que ocorra vai desviar para o `except` e dar a mesma mensagem " *Não é possível calcular a divisão*", independente de qual erro realmente ocorreu: se foi na leitura de um dado (`ValueError`) ou se foi divisão por zero (`ZeroDivisionError`). Para programas profissionais isso não é nada adequado.

Exemplo 6.3 - adaptado

```
try:
    A = int(input('Digite A: '))
    B = int(input('Digite B: '))
    R = A / B
    print(f'Resultado = {R}')
except:
    print('Não é possível calcular a divisão')
```

primeira execução

```
Digite A: 20
Digite B: 0
Não é possível calcular a divisão
```

segunda execução

```
Digite A: 5o ← Foi digitada a letra 'o' ao invés de zero (0)
Não é possível calcular a divisão
```

Para promover um tratamento adequado a cada situação pode-se identificar a classe da exceção na cláusula `except` conforme mostrado no exemplo 6.4. Nesse exemplo foram usados dois `excepts` nomeados e um genérico. Quando ocorre um erro o interpretador Python primeiro procura a existência de um `except` nomeado e se encontrar desvia a execução para ele; caso contrário ele executa o `except` genérico (não nomeado). Só pode haver um `except` genérico.

Exemplo 6.4



```
try:
    A = int(input('Digite A: '))
    B = int(input('Digite B: '))
    R = A / B
    print(f'Resultado = {R}')
except ZeroDivisionError: # except nomeado
    print('B não pode ser zero')
except ValueError:       # except nomeado
    print('Digite números inteiros para A e B')
except:                  # except genérico (não-nomeado)
    print('Não é possível calcular a divisão. Erro desconhecido')
```

primeira execução

```
Digite A: 20
Digite B: 0
B não pode ser zero
```

segunda execução

```
Digite A: texto
Digite números inteiros para A e B
```

terceira execução

```
Digite A: 18
Digite B: 4
Resultado = 4.5
```

Na primeira execução do exemplo 6.4 foi fornecido 0 para B, ocorreu o erro e a execução desviou para o `except ZeroDivisionError`. Na segunda execução foi fornecido um texto na leitura de A, ocorreu o

erro e a execução desviou para o `except ValueError`. Na terceira execução foram fornecidos valores adequados para A e B e não ocorreu nenhum erro.

6.3 TRATAMENTO DE EXCEÇÕES EM PYTHON – FORMA COMPLETA

O comando `try` tem outros dois blocos ainda não mencionados e serão vistos agora. As linhas abaixo mostram sua estrutura completa

```
try:
    <bloco 1>
except:
    <bloco 2>
else:
    <bloco 3>
finally:
    <bloco 4>
```

Como se pode ver, o comando `try`, além dos vários blocos `except`, contém outras duas cláusulas opcionais: `else` e `finally`. Essas cláusulas devem ser colocadas nessa ordem.

Quando o bloco 1 é executado ele está protegido. Supondo que nenhum erro ocorra, imediatamente após seu término é feita a execução do bloco 3 que está no `else`. Se alguma exceção ocorrer o código de tratamento no `except` é executado e o código do `else` não é executado.

A outra parte opcional `finally` sempre é executada, ocorra erro ou não. Se uma cláusula `finally` estiver presente, ela será executada como a última tarefa antes da conclusão da instrução `try`. Se ambos, `else` e `finally`, estiverem presentes o `else` será executado antes do `finally`.

O objetivo da cláusula `finally` é permitir a implementação de ações de finalização e limpeza, que eventualmente sejam necessárias independentemente da ocorrência de exceções, por exemplo: eliminação de arquivos temporários; fechamento de conexão com o banco de dados; ou a liberação de recursos de rede.

Este assunto de tratamento de exceções ainda não está encerrado.
Aqui temos o essencial para este momento do curso.
Há mais para saber, porém é conteúdo para o Módulo Intermediário.

Mais sobre exceções você encontra neste link:
<https://docs.python.org/pt-br/3/tutorial/errors.html>

Capítulo 7

OBJETOS COMPOSTOS DE PYTHON – OBJETOS SEQUENCIAIS

Os objetos compostos da linguagem Python constituem recursos muito valiosos aos programadores. Permitem capacidade de processamento de grandes volumes de dados, com eficiência, confiabilidade e flexibilidade. São elementos fundamentais que você, estudante, deve se dedicar a aprender.

Estes objetos compostos também são chamados de objetos estruturados pois seu conteúdo é formado por elementos que podem ser acessados individualmente e também em grupo.

No universo de objetos estruturados de Python há três que são conhecidos como Objetos Sequenciais:

- os strings - classe `str` (já temos trabalhado com eles);
- as listas - classe `list`;
- as tuplas - classe `tuple`;

A principal característica dos tipos sequenciais é que seus elementos são mantidos em uma organização baseada em um índice numérico crescente da esquerda para a direita, que começa em zero e sofre o incremento de um a um. Com isso, é possível ao programador acessar individualmente seus elementos através do uso de índices especificados entre colchetes: `[]`.

Vamos abordar primeiro a classe `list`, que é um dos elementos mais importantes de Python.

7.1 LISTAS – CLASSE `list`

As listas foram brevemente mencionadas no quadro 2.2 do capítulo 2 e agora é hora de nos aprofundarmos neste assunto.

A classe `list` é uma sequência. Listas são usadas para armazenar objetos de qualquer classe. Observe a figura 7.1 e o exemplo 7.1.

Figura 7.1 – Lista com 10 elementos

36	25	21	48	17	9	16	23	29	31
0	1	2	3	4	5	6	7	8	9

fonte: o Autor

Na figura 7.1 nós temos uma representação gráfica do que seria uma lista em Python: um grupo de objetos reunidos em um mesmo identificador (nome) e que podem ser acessados individualmente através de um índice numérico. Esse índice é representado pelos números de 0 a 9 na parte de baixo da imagem. Os números dentro das caixas representam o conteúdo de cada objeto da lista.

Exemplo 7.1



```
L = [36, 25, 21, 48, 17, 9, 16, 23, 29, 31]
print('Exibição da lista completa')
print(L)
print('\nExibição dos elementos individuais')
i = 0
while i < len(L):
    print(L[i], end=' ')
    i = i + 1
print('\nFim do Programa')
Exibição da lista completa
[36, 25, 21, 48, 17, 9, 16, 23, 29, 31]

Exibição dos elementos individuais
36 25 21 48 17 9 16 23 29 31
Fim do Programa
```

Em um programa Python poderíamos fazer essa lista da forma mostrada no exemplo 7.1, onde a lista `L` é pré-carregada com os valores da figura 7.1 e em seguida exibida de duas formas:

- primeira forma: exibição da lista como um todo com a função `print(L)`;
- segunda forma: através do laço de repetição usado para exibir um elemento por vez usando o índice do elemento `print(L[i])`;

Vamos explorar um pouco mais usando o ambiente Idle para interagir com a lista. No exemplo 7.2 criamos a lista com 10 elementos; exibimos o tipo de `L`; exibimos individualmente alguns elementos usando o índice; exibimos o tamanho da lista com o uso da função `len()`; alteramos o primeiro valor da lista; e por fim exibimos a lista como um todo para conferir se a alteração do primeiro item de fato ocorreu.

Exemplo 7.2



```
>>> L = [36, 25, 21, 48, 17, 9, 16, 23, 29, 31]
>>> type(L)      # exibição da classe de L
<class 'list'>
>>> L[0]         # exibição do primeiro elemento -> L[0]
36
>>> type(L[0])   # exibição da classe do primeiro elemento
<class 'int'>
>>> L[9]         # exibição do último elemento -> L[9]
31
>>> len(L)       # a função len() retorna o tamanho da lista
10
>>> L[0] = 999   # alteração do primeiro elemento da lista
>>> print(L)
[999, 25, 21, 48, 17, 9, 16, 23, 29, 31]
(exemplo interativo feito com IDE Idle)
```

Nesta lista todos os elementos são da classe `int`, de modo que dizemos que essa é uma lista homogênea. Nada nos impede de criarmos listas com objetos de diferentes classes.

É comum que todos os elementos da lista sejam da mesma classe e quando isso ocorre dizemos que a lista é homogênea. Porém isso não é obrigatório, ou seja, uma lista pode conter elementos classes diferentes e aí dizemos que a lista é heterogênea.

7.2 OPERAÇÕES COM LISTAS

As listas de Python oferecem ao programador uma gama muito grande de operações que podem ser realizadas. Essas operações podem ser feitas tanto a partir de funções de Python externas à classe `list` quanto a partir dos métodos disponíveis na própria classe `list`. Nesta seção vamos nos dedicar a conhecer essas operações

7.2.1 CRIAÇÃO DE UMA LISTA DO ZERO

Por ser muito frequente, a primeira operação que precisamos aprender é como criar uma lista vazia e incluir elementos. O exemplo 7.3 mostra que para criar a lista basta executar a atribuição `L = []`, que pode ser lida como "objeto recebe par de colchetes". Usando a função `type()` conferimos que `L` foi criado e que ele é da classe `list`. Em seguida, usando o método `.append()` fizemos a inclusão de dois números reais e exibimos a lista com `print()`. O método `.append()` sempre faz a inclusão de objetos no final da lista.

Exemplo 7.3

[Teste este código no Idle](#)

```
>>> L = []
>>> type(L)
<class 'list'>
>>> L.append(3.88)
>>> L.append(17.5)
>>> print(L)
[3.88, 17.5]
>>> type(L[0])
<class 'float'>
```

(exemplo interativo feito com IDE Idle)

Tarefa: Abra o Idle e execute esses exemplos.

7.2.2 FORMAS DE INDEXAÇÃO

Indexação de uma lista é o uso de um valor dentro dos colchetes para acessar um elemento individual da lista. O primeiro elemento da sequência sempre terá índice 0 (erro). O índice deve sempre ser um inteiro na forma literal, um objeto `int`, ou uma fórmula que resulte em valor inteiro, como mostrado a seguir.

Exemplo 7.4

[Teste este código no Idle](#)

```
>>> A = [10, 12, 14, 16]
>>> i = 0
>>> print(A[i])
10
>>> print(A[i+1])
12
>>> print(A[i+2])
14
```

(exemplo interativo feito com IDE Idle)

Em Python é possível fazer a indexação com números negativos, que permitirá o acesso aos elementos de trás para frente, sendo que o índice `-1` se refere aos últimos objeto da lista.

Exemplo 7.5

Teste este código no Idle

```
>>> A = [10, 12, 14, 16]
>>> print(A[-1])      # último elemento da lista
16
>>> print(A[-2])      # penúltimo elemento da lista
14
>>> print(A[-3])      # e assim por diante, de trás para frente
12
>>> print(A[-4])
10
```

(exemplo interativo feito com IDE Idle)

O Python não permite indexação fora dos limites. Se houver a tentativa de usar um índice de elemento (positivos ou negativos) que não existe uma exceção `IndexError` será gerada.

Exemplo 7.6

Teste este código no Idle

```
>>> A = [10, 12, 14, 16]
>>> print(A[9])
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    print(A[9])
IndexError: list index out of range
```

(exemplo interativo feito com IDE Idle)

Tarefa: Abra o Idle e execute esses exemplos.

7.2.3 ELIMINAÇÃO DE ELEMENTOS USANDO A FUNÇÃO `del`

Muitas vezes há necessidade de eliminar um ou mais elementos de uma lista. Um dos modos de fazer isso é usando a função `del`. Para usá-la é preciso saber o índice do elemento você quer eliminar e escrevê-la no formato mostrado no exemplo 7.7.

Exemplo 7.7

Teste este código no Idle

```
>>> A = [10, 12, 14, 16]
>>> del A[0]          # elimina o primeiro elemento da lista A
>>> print(A)
[12, 14, 16] # o valor 10 que estava na primeira posição foi eliminado
```

(exemplo interativo feito com IDE Idle)

7.2.4 FATIAMENTO DE LISTAS

O fatiamento (em inglês: slicing) é uma operação usada com frequência em Python aplicável a listas, tuplas e strings. Essa operação permite extrair parte da sequência de origem para produzir outra sequência. A construção de um fatiamento é feita como indicado na linha abaixo:

```
Destino = Origem[ini:fim]
```

Com um código assim a lista `Destino` será criada com os elementos da lista `Origem`, começando na posição `ini` e terminando na posição `fim-1`. O elemento da posição `fim` nunca é incluído.

No exemplo 7.8 a lista `Origem` tem 10 elementos e é feito o fatiamento `[3:6]` de modo que os elementos das posições 3, 4 e 5 serão incluídos na lista `Destino1`. O elemento da posição 6 ficará de fora – esteja sempre atento a isso.

Adicionalmente o fatiamento pode conter um terceiro parâmetro, da seguinte forma:

```
Destino = Origem[ini:fim:passo]
```

Neste caso o terceiro parâmetro é o passo, que será usado como um "pulo" entre elementos sucessivos. No segundo caso do exemplo foi usado o fatiamento `[1:7:2]` que resulta em seleção dos elementos das posições 1, 3 e 5, ou seja `[25, 48, 9]`. Por padrão o passo é 1 quando não especificado.

Também é possível omitir os parâmetros `ini` e `fim`.

```
Destino = Origem[:fim]
```

No caso da omissão de `ini` o fatiamento é feito a partir do início da lista e é terminado em `fim-1`.

```
Destino = Origem[ini:]
```

No caso da omissão de `fim` o fatiamento começará em `ini` e se estenderá até o final da lista.

Exemplo 7.8



```
>>> Origem = [36, 25, 21, 48, 17, 9, 16, 23, 29, 31]
>>> Destino1 = Origem[3:6]          # primeiro caso
>>> print(Destino1)
[48, 17, 9]
>>> Destino2 = Origem[1:7:2]        # segundo caso - incluindo o passo
>>> print(Destino2)
[25, 48, 9]
>>> Destino3 = Origem[:4]           # terceiro caso - omitindo ini
>>> print(Destino3)
[36, 25, 21, 48]
>>> Destino4 = Origem[6:]           # quarto caso - omitindo fim
>>> print(Destino4)
[16, 23, 29, 31]
```

(exemplo interativo feito com IDE Idle)

Quadro 7.1 – Resumo do fatiamento

Forma	Interpretação
<code>Lst[ini:fim]</code>	O fatiamento começa em <code>ini</code> e termina em <code>fim - 1</code>
<code>Lst[:fim]</code>	O fatiamento começa no elemento 0 e termina em <code>fim - 1</code>
<code>Lst[ini:]</code>	O fatiamento começa em <code>ini</code> e vai até o final da lista
<code>Lst[:]</code>	O fatiamento engloba a lista inteira
<code>Lst[ini:fim:p]</code>	O fatiamento começa em <code>ini</code> e termina em <code>fim - 1</code> e tem passo <code>p</code>
<code>Lst[:fim:p]</code>	O fatiamento começa no elemento 0 e termina em <code>fim - 1</code> e tem passo <code>p</code>
<code>Lst[ini::p]</code>	O fatiamento começa em <code>ini</code> e vai até o final da lista e tem passo <code>p</code>

A forma de fatiamento sem parâmetros `Destino = Origem[:]` é de grande interesse e é detalhada na próxima seção.

7.2.5 CÓPIA DE UMA LISTA USANDO FATIAMENTO

Copiar uma lista é uma operação frequentemente necessária. Para programadores desavisados e iniciantes em Python essa operação pode ser uma armadilha. Veja as linhas a seguir e reflita: ao fazer `B = A` a lista A foi copiada na lista B?

```
A = [10, 12, 14, 16]
B = A
```

A resposta é **não, a lista B não é uma cópia de A**. Veja o exemplo 7.9. Depois de fazer `B = A` mandamos exibir o `id` dos dois objetos e constatamos que eles são iguais.

Exemplo 7.9

```
>>> A = [10, 12, 14, 16]
>>> B = A
>>> id(A)
1837485118208
>>> id(B)
1837485118208
>>> B[0] = 99
>>> print(B)
[99, 12, 14, 16]
>>> print(A)
[99, 12, 14, 16]
```

(exemplo interativo feito com IDE Idle)

Isso acontece porque a classe `list` de Python é mutável. Retorne à figura 2.2 e releia a seção 2.3 sobre o Modelo de Dados de Python para relembrar os conceitos de classe imutável e classe mutável.

Quando fizemos `B = A` neste exemplo o que o Python fez foi criar o objeto B em memória e copiar o `id` de A para B. Com isso, na prática, só existe uma lista em memória e os dois objetos, A e B, apontam para ela. Por consequência qualquer alteração de dados que for feita em B também refletirá em A e vice-versa. No exemplo alteramos o primeiro elemento de B para 99 e mostramos que A também ficou alterado.

Para quem está aprendendo Python isso pode ser estranho e incômodo no início, então fique atento e não caia nessa armadilha. Agora a pergunta: se não dá para fazer `B = A`, como fazer então?

A resposta é simples: use o fatiamento para copiar todos os elementos para uma nova lista, assim:

Exemplo 7.9 – alterado

```
>>> A = [10, 12, 14, 16]
>>> B = A[:] # fatiamento que pega todos os elementos de A e copia em B
>>> id(A)
1837485118208
>>> id(B)
1837529145216
>>> B[0] = 99
>>> print(B)
[99, 12, 14, 16]
>>> print(A)
[10, 12, 14, 16]
```

(exemplo interativo feito com IDE Idle)

Nessa nova versão do exemplo 7.9 foi usado o fatiamento no formato `Destino = Origem[:]` que faz com que a lista `Destino` seja uma cópia integral e com `id` próprio da lista `Origem`.

Aviso e Dica

O aviso: cuidado com cópias de listas em seus programas. Se elas forem pequenas, certamente não haverá problemas. Porém, se suas listas forem grandes, com muitos de milhares de elementos, fazer cópias e mais cópias pode levar a uma lentidão indesejada na execução do seu programa. Assim, a dica é: só faça cópia de uma lista muito grande quando isso for absolutamente necessário e saiba que isso é necessário em poucas ocasiões. Pense a respeito...

7.2.6 MÉTODOS DA CLASSE `list`

A classe `list` tem um bom conjunto de métodos que nos auxiliam a manipular seus elementos. O quadro 7.2 mostra todos eles.

Quadro 7.2 – Métodos da classe `list`

Método	Descrição
<code>.append(object)</code>	Acrescenta um objeto ao final da lista ex. <code>L.append(5)</code>
<code>.clear()</code>	Limpa a lista, removendo todos seus elementos ex. <code>L.clear()</code>
<code>.copy()</code>	Produz uma cópia da lista <code>L</code> . Usar este método é uma alternativa ao uso do fatiamento mostrado na seção anterior ex. <code>NovaLista = L.copy()</code>
<code>.count(object)</code>	Retorna o número de ocorrências de um objeto dentro da lista ex. <code>Qtde = L.count(5)</code>
<code>.extend(iterable)</code>	Expande a lista acrescentando a ela todos os elementos contidos no objeto <code>iterable</code> passado como parâmetro ex. <code>L.extend(OutraLista)</code>
<code>.index(value, [start, stop])</code>	Retorna o índice da primeira ocorrência do objeto <code>value</code> dentro da lista. Se <code>start</code> e <code>stop</code> (opcionais) forem fornecidos o método considera apenas seu intervalo. Caso <code>value</code> não esteja na lista é gerado um erro ex. <code>posição = L.index(5)</code>
<code>.insert(index, object)</code>	Insere o objeto fornecido na posição dada por <code>index</code> , deslocando todos os demais para a direita ex. <code>L.insert(2, 30)</code>
<code>.pop(index)</code>	Retorna o elemento que está na posição dada por <code>index</code> e o remove da lista. ex. <code>L.pop(0)</code>
<code>.remove(value)</code>	Remove da lista a primeira ocorrência de <code>value</code> . Se o valor não estiver na lista gera um erro. ex. <code>L.remove(5)</code>
<code>.reverse()</code>	Inverte a posição dos elementos dentro da lista, o primeiro valor passa a ser o último, o segundo passa a penúltimo e assim por diante. Não retorna nada pois inverte a própria lista. ex. <code>L.reverse()</code>
<code>.sort(...)</code>	Ordena a lista, colocando-a em ordem crescente ou decrescente. Não retorna nada, pois ordena a própria lista. ex. <code>L.sort()</code> # ordem crescente <code>L.sort(reverse=True)</code> # ordem decrescente Observação: 1. Cuidado ao usar o método <code>.sort()</code> com listas heterogêneas. Se o interpretador Python não conseguir definir a forma de ordenação entre os vários elementos ele irá falhar.

Exemplo 7.10



Este exemplo é muito longo, pois mostra os métodos da classe `list`. Então seu código não foi colocado aqui. O vídeo está disponível.

(exemplo interativo feito com IDE Idle)

7.2.7 OPERADOR `in`

O operador `in` permite ao programador verificar se um valor está presente em uma lista.

Alternativamente pode-se usá-lo na forma `not in` para verificar se um valor não está na lista.

Este é um operador de uso geral em Python. Portanto, seu uso não é restrito às listas, podendo ser usado com a maioria das classes de objetos compostos. O Exemplo 7.2 ilustra seu uso com uma lista.

Exemplo 7.11

Teste este código no Idle

```
>>> L = [36, 25, 21, 48, 17, 9, 16, 23, 29, 31]
>>> print(25 in L)
True # 25 está em L, então in retornou True
>>> print(99 in L)
False # 99 não está em L, então in retornou False
>>> print(25 not in L)
False # 25 está em L, então not in retornou False
>>> print(99 not in L)
True # 99 não está em L, então not in retornou True
(exemplo interativo feito com IDE Idle)
```

Agora vamos fazer um exercício completo usando uma lista. Veja o exercício resolvido 7.1

Exercício Resolvido 7.1



Enunciado: Escreva um programa que leia três números inteiros: o primeiro termo P , a razão R e a quantidade Q de termos de uma progressão aritmética. O programa deve calcular os Q termos da progressão colocando-os em uma lista e no final exibi-la na tela.

obs: esse problema já foi abordado, sem o uso de listas, no exercício resolvido 5.3.

```
P = int(input("Digite o primeiro termo: "))
R = int(input("Digite a razão: "))
Q = int(input("Digite a qtde de termos: "))
Termos = [P] # cria a lista já com o primeiro termo
i = 0
while i < Q-1: # vai até Q-1 porque o primeiro termo já esta na lista
    P = P + R # calcula o próximo termo
    Termos.append(P) # acrescenta o termo à lista
    i += 1
print('Lista resultante')
print(Termos)
print('Fim do Programa')
Digite o primeiro termo: 4
Digite a razão: 7
Digite a qtde de termos: 12
Lista resultante
[4, 11, 18, 25, 32, 39, 46, 53, 60, 67, 74, 81]
Fim do Programa
```

Nesta solução a lista `Termos` é inicialmente criada contendo o primeiro termo. A cada repetição do laço `while` um novo termo é calculado e acrescentado na lista. Para que a lista termine com a quantidade correta Q de termos é preciso tomar um cuidado. Como o primeiro termo é pré-colocado na lista fora do laço, na condição usada no `while` foi preciso descontar 1 de Q . Se isso não fosse feito a lista ficaria no final com um termo a mais do que foi pedido.

Não-Pythônico e Pythônico

Na comunidade Python esses dois termos costumam aparecer com frequência. Vamos entender isso.

Quando dizemos que algo é Pythônico estamos nos referindo à forma ideal de como um programa deve ser escrito usando Python. "ser Pythônico" não é uma definição exata. Na verdade, ela pode ser um tanto vaga e sujeita a interpretações, uma vez que sempre existe mais de um modo de se fazer algo.

O importante é você saber que isso existe e também saber que os códigos que foram apresentados até o momento seriam considerados Não-Pythônicos por programadores experientes na linguagem. Com certeza, um programador Python diria que o exercício resolvido 7.1 é totalmente Não-Pythônico.

Isso não chega a ser um problema, pois estamos apenas começando. Para escrever programas Pythônicos precisamos de mais conhecimentos específicos sobre a linguagem Python.

Então, o que devemos fazer é seguir no aprendizado. Vamos adiante.

7.3 A CLASSE `range`

Muitos textos sobre a linguagem Python fazem referência a `range` como sendo uma função, quando na verdade, segundo a documentação oficial – seção 4.6.6, do capítulo 4 da "The Python Standard Library" – trata-se de um tipo sequencial imutável. Por este motivo, neste texto será usada a expressão tipo `range` ao invés de função `range`.

Já mencionada no quadro 2.2 a classe `range` existe para produzir um objeto imutável que é uma sequência de números inteiros. Ela tem grande importância na implementação de vários tipos de algoritmos e seu uso é muito simples. Sua forma geral é esta:

```
range(start, stop[, step])
```

A sequência começará com o valor `start` e terminará com o valor `stop - 1`. Assim como no fatiamento, o valor final nunca é incluído. Se o objeto `step` for fornecido seu valor será usado como passo para construção da sequência. Lembre-se sempre desses detalhes:

- O parâmetro `stop` é obrigatório.
- O parâmetro `start` é opcional e se não for fornecido será assumido o valor padrão 0.
- O parâmetro `step` é opcional e se não for fornecido será assumido o valor padrão 1.

Alguns exemplos:

```
range(10)      gera a sequência [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(3, 8)    gera a sequência [3, 4, 5, 6, 7]
range(5, 12, 3) gera a sequência [5, 8, 11]
```

Veja agora o exercício resolvido 7.2.

Exercício Resolvido 7.2



Enunciado: Usando a classe `range`, escreva um programa que leia três números inteiros: o primeiro termo P , a razão R e a quantidade Q de termos de uma progressão aritmética. O programa deve calcular os Q termos da progressão colocando-os em uma lista e no final exibi-la na tela.

obs: é o mesmo enunciado do exercício resolvido anterior 7.1

```
P = int(input("Digite o primeiro termo: "))
R = int(input("Digite a razão: "))
Q = int(input("Digite a qtde de termos: "))
ultimo = P + R * (Q-1)
Termos = list(range(P, ultimo+1, R)) # a classe range gera a PA
print('Lista gerada com range')
print(Termos)
print('Fim do Programa')
```

Digite o primeiro termo: 4
Digite a razão: 7
Digite a qtde de termos: 12
Lista gerada com range
[4, 11, 18, 25, 32, 39, 46, 53, 60, 67, 74, 81]
Fim do Programa

Não-Pythônico e Pythônico

Aqui temos o mesmo problema com outra solução. Esta solução 7.2 pode ser considerada mais Pythônica do que a solução 7.1. Motivo: ela utiliza recurso específico de Python que elimina a necessidade do laço `while`, sendo mais compacta e legível. É disso que se trata quando mencionamos Não-Pythônico e Pythônico.

Neste exercício temos o mesmo enunciado do exercício resolvido 7.1, porém a solução é diferente. Nela foi usada classe `range` para gerar a PA. Note que para usar `range` precisamos saber o último termo da PA desejada. Assim foi calculado:

```
ultimo = P + R * (Q-1)
```

Essa fórmula é decorrente da matemática, não exatamente um assunto de programação de computadores. Precisamos desse último termo se quisermos usar `range`, então fomos buscar na matemática a forma de calculá-lo. Isso ocorre sempre em programação. Muitas vezes precisamos de informações especializadas da área de conhecimento para a qual estamos escrevendo o programa.

No que diz respeito ao uso do `range`, o código usado requer duas explicações.

```
Termos = list(range(P, ultimo+1, R))
```

Primeiro, note que no parâmetro `stop` foi usado `ultimo+1`. Isso é necessário pois o `stop` nunca está incluído na sequência gerada, então para garantir a inclusão do `ultimo` precisamos estabelecer `ultimo+1` como `stop` do `range`.

Segundo, note que a classe `range` foi colocada dentro da função de conversão `list`:

```
... list(range(...))
```

Essa conversão da classe `range` para a classe `list` é necessária para que possamos ver os valores da sequência na tela. Se não for feita vamos o que está a seguir:

```
Termos = range(P, ultimo+1, R) # sem usar list()
print(Termos)
range(7, 52, 4)                # é isso que vamos ter na tela
```

7.4 O COMANDO `for`

7.4.1 CONCEITO

Já foi mencionado no início do capítulo 5 que o comando `for` é um comando de repetição. Portanto, ele serve para a criação de laços de repetição, mas ele é bem diferente do comando `while`. Podemos dizer que o `while` é um comando de repetição de uso geral, enquanto o `for` tem um uso bem específico.

O comando `for` é usado para iterar sobre os elementos de qualquer objeto estruturado de Python, tais como: listas, strings, tuplas, dicionários, conjuntos, etc.

Iterar e iteração
Acostume-se com esses termos.

Iterar é
percorrer os elementos de uma sequência,
na ordem em que eles estão dentro dela e
realizar um processamento com cada um.

Iteração é o processo de iterar.

Veja seu uso no exemplo 7.12. Neste exemplo uma lista com 4 números inteiros é criada e em seguida o comando `for` é usado para iterar sobre ela. Para tornar a iteração possível o objeto `valor` tem papel preponderante. Ele recebe cada elemento contido na lista `L`, um por vez, ou seja, um objeto de `L` a cada repetição. E dentro do laço `valor` pode ser usado normalmente para qualquer fim que o programador necessite.

Exemplo 7.12



```
L = [21, 45, 17, 28]
pos = 0
for valor in L:
    print(f'A posição {pos} contém {valor}')
    pos += 1
print('Fim do Programa')
A posição 0 contém 21
A posição 1 contém 45
A posição 2 contém 17
A posição 3 contém 28
Fim do Programa
```

A descrição formal do funcionamento do comando `for` é a seguinte:

```
for objctrl in objseq:
    bloco1
else:
    bloco2
```

- O objeto `objseq` é avaliado para verificar se é um objeto válido para o comando `for`. Se não for válido um erro é gerado e o processo termina; se for válido avança para o próximo passo;
- O objeto `objctrl` recebe o primeiro elemento de `objseq` e o `bloco1` é executado uma vez;
- Ao término da execução do `bloco1` o `objctrl` recebe o próximo elemento de `objseq` e o `bloco1` é executado novamente. Isto se repete para cada item presente em `objseq`.

Caso o laço de repetição termine normalmente o `bloco2` dentro o `else` é executado.

Os comandos `continue` e `break`, vistos quando falamos do comando `while`, também se aplicam ao comando `for`. Veja as seções 5.3.1 e 5.2.2 para relembrar sobre eles.

- O comando `continue` executado no `bloco1` interrompe a atual repetição e continua com o próximo item;
- O comando `break` executado no `bloco1` termina o laço de repetição. Se isto ocorrer o `bloco2` dentro da cláusula `else` não será executado.

7.4.2 USO COMBINADO DE `range` E `for`

É muito comum usar o comando `for` combinado com a classe `range`. O exemplo 7.13 mostra um caso simples. Adiante neste curso essa combinação vai ocorrer muitas vezes.

Exemplo 7.13

[Teste este código no PyCharm](#)

```
for i in range(4):
    print(f'Valor i da vez = {i}')
print('Fim do Programa')
Valor i da vez = 0
Valor i da vez = 1
Valor i da vez = 2
Valor i da vez = 3
Fim do Programa
```

Laços com `while` são genéricos e sempre é possível usá-los.
Laços com `for` são específicos para iterações com objetos estruturados.

Nem sempre é possível usar um laço `for`.

Recomendação:
Sempre que possível use um laço `for`.

Por quê?
Não haverá necessidade de usar objetos extras para controle do laço.
Seu código ficará mais compacto.
Seu código ficará mais legível.

7.5 OPERADORES DE CONCATENAÇÃO E MULTIPLICAÇÃO APLICADO A CLASSES SEQUENCIAIS

Nesta seção vamos apresentar dois operadores muito simples e práticos que podem ser usados com listas, tuplas e strings.

7.5.1 CONCATENAÇÃO COM "+"

Considere ao exemplo 7.14 a seguir.

Exemplo 7.14

[Teste este código no Idle](#)

```
>>> A = [10, 12, 14, 16]
>>> B = [20, 22, 24]
>>> R = A + B      # operador de concatenação '+' usado para juntar duas listas
>>> print(R)
[10, 12, 14, 16, 20, 22, 24]
```

(exemplo interativo feito com IDE Idle)

Neste exemplo são definidas duas listas A e B, e em seguida foi usado o operador "+" para juntar as duas listas. Assim, no contexto de classes sequenciais o operador "+" é denominado "operador de concatenação".

Ele também pode ser usado com tuplas e strings.

7.5.2 MULTIPLICAÇÃO COM "*"

Considere ao exemplo 7.15 a seguir.

Exemplo 7.15

[Teste este código no Idle](#)

```
>>> A = [1, 2, 3] * 3
>>> print(A)
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> B = [0] * 10
>>> print(B)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

(exemplo interativo feito com IDE Idle)

O operador de multiplicação "*" serve para criar sequências com múltiplas cópias de uma subsequência. Ele é particularmente útil nas situações em que precisamos de uma lista com uma certa quantidade pré-definida de elementos e carregados com um determinado valor inicial.

7.6 EXERCÍCIOS USANDO LISTAS

Agora é hora de praticar. Sugerimos que assista aos vídeos, implemente os exercícios resolvidos e faça exercícios propostos.

Exercício Resolvido 7.3



Enunciado: Escreva um programa que leia um número inteiro *N*. Em seguida leia *N* números reais carregando-os em uma lista. Ao final exiba os elementos da lista na tela, sendo um em cada linha

```
N = int(input('Digite a quantidade: '))
L = []
for i in range(N):
    X = float(input(f' elemento {i}: '))
    L.append(X)
print('\nResultado:')
for valor in L:
    print(f' {valor}')
print('Fim do Programa')
```

```
Digite a quantidade: 4
 elemento 0: 3.1
 elemento 1: 19.8
 elemento 2: 7.35
 elemento 3: 12.4
```

```
Resultado:
3.1
19.8
7.35
12.4
```

```
Fim do Programa
```

Nesta solução 7.3 o comando `for` foi usado duas vezes. Na primeira, em combinação com a classe `range`, são lidos vários valores de entrada e cada um é adicionado à lista `L`. Terminada a fase de aquisição dos dados é feita a exibição com o segundo comando `for`.

Exercício Proposto 7.1

Enunciado: Altere a solução do ex.resolvido 7.3 para exibir os números reais da lista com duas casas decimais.

Exercício Proposto 7.2

Enunciado: Altere a solução do ex.resolvido 7.3 incluindo o comando `try-except` na leitura dos números reais para evitar a digitação incorreta dos valores. Quando ocorrer uma exceção uma mensagem deve ser exibida na tela informando esta condição.

Dica: Relembre o tratamento de exceções consultando o capítulo 6, em especial o exemplo 6.4

Exercício Proposto 7.3

Enunciado: Altere a solução do ex.resolvido 7.3 para exibir os resultados em ordem inversa à ordem de leitura

Dica: Aplique o método `.reverse()` apresentado no quadro 7.2 e visto no vídeo do exemplo 7.10

Exercício Proposto 7.4

Enunciado: *Altere a solução ex.resolvido 7.3 para exibir os resultados em ordem crescente*

Dica: *Aplique o método `.sort()` apresentado no quadro 7.2 e visto no vídeo do exemplo 7.10*

Exercício Resolvido 7.4

Enunciado: *Escreva um programa que leia um número inteiro N. Em seguida leia N números inteiros carregando os valores negativos em uma lista e os positivos em outra. Ao final exiba na tela cada uma das listas juntamente com seu tamanho.*

obs. Se o valor zero for fornecido ele deve ser incluído na lista de positivos.

```
N = int(input('Digite a quantidade: '))
LstNeg = []
LstPos = []
for i in range(N):
    X = int(input(f' elemento {i}: '))
    if X >= 0:
        LstPos.append(X)
    else:
        LstNeg.append(X)
print(f'\nLista de negativos: {LstNeg}, contém {len(LstNeg)} elementos')
print(f'\nLista de positivos: {LstPos}, contém {len(LstPos)} elementos')
print(f'\nFim do Programa')
```

Digite a quantidade: 5

elemento 0: 184

elemento 1: -93

elemento 2: -12

elemento 3: 57

elemento 4: -86

Lista de negativos: [-93, -12, -86], contém 3 elementos

Lista de positivos: [184, 57], contém 2 elementos

Fim do Programa

Exercício Proposto 7.5

Enunciado: *Responda a seguinte questão. A inicialização das listas foi feita da seguinte forma:*

`LstNeg = []`

`LstPos = []`

por quê a solução 7.4 ficaria errada se a inicialização das listas fosse feita assim?

`LstNeg = LstPos = []`

Justifique sua resposta.

Dicas: *a) altere o programa do exercício resolvido 7.4 e teste para ver o que acontece;*

b) releia as seções 2.3, 2.4 e 7.2.5

Exercício Proposto 7.6

Enunciado: *Altere a solução ex.resolvido 7.4 para exibir as listas em ordem crescente*

Exercício Resolvido 7.5



Enunciado: Escreva um programa que permaneça em laço lendo números inteiros. O laço termina quando for digitado 0 (zero). Cada valor diferente de zero digitado deve ser colocado em uma lista. Ao final exiba a lista na tela e a quantidade de elementos que ela contém.

```
LstValores = []
valor = int(input('Digite um inteiro: '))
while valor != 0:
    LstValores.append(valor)
    valor = int(input('Digite um inteiro: '))
print('\nResultado')
print(LstValores)
print(f'A lista contém {len(LstValores)} elementos')
print('Fim do Programa')

Digite um inteiro: 138
Digite um inteiro: -79
Digite um inteiro: -14
Digite um inteiro: 63
Digite um inteiro: -49
Digite um inteiro: 41
Digite um inteiro: 0

Resultado
[138, -79, -14, 63, -49, 41]
A lista contém 6 elementos
Fim do Programa
```

Perceba que nesta solução foi usado o comando `while` ao invés do `for`. O motivo é o fato de que não sabemos quantas vezes o laço irá se repetir. Como o enunciado impõe que o laço só termina quando o dado de entrada for 0 (zero), a permanência das repetições depende daquilo que o usuário digita. Este é um exemplo de situação em que não é possível usar o comando `for`.

Outra restrição do enunciado é: "cada valor diferente de zero deve ser colocado em uma lista", ou seja, o zero não pode entrar na lista. Para lidar com esta restrição foi usada a técnica de "aquisição do dado na saída do laço". Isso é um aspecto de lógica que se aplica a qualquer linguagem. Para implementar essa técnica fazemos a leitura do primeiro dado antes do laço começar; dentro do laço o primeiro dado é processado e a última tarefa dentro do laço é ler o próximo dado – se esse dado lido for zero o laço termina, mas se não for o laço continua e já podemos processar o novo dado. A alternativa a isso seria o código abaixo:

```
valor = 1
while valor != 0:
    valor = int(input('Digite um inteiro: '))
    if valor != 0:
        LstValores.append(valor)
```

Nesta alternativa temos que garantir que o laço inicie, para isso fizemos `valor = 1` e dentro do laço temos que acrescentar um `if` para garantir que o zero não entre na lista.

Exercício Resolvido 7.6

Enunciado: Escreva um programa que permaneça em laço lendo números inteiros. O laço termina quando for digitado 0 (zero). Cada valor diferente de zero digitado deve ser colocado em uma lista, desde que ele ainda não esteja lá, ou seja, valores repetidos não são aceitos. Se um valor repetido for digitado, o programa deve exibir uma mensagem na tela.

Ao final exiba a lista e a quantidade de elementos que ela contém.

```
LstValores = []
valor = int(input('Digite um inteiro: '))
while valor != 0:
    if valor not in LstValores:
        LstValores.append(valor)
    else:
        print(f' erro! o valor {valor} já está na lista')
    valor = int(input('Digite um inteiro: '))
print('\nResultado')
print(LstValores)
print(f'A lista contém {len(LstValores)} elementos')
print('Fim do Programa')

Digite um inteiro: 23
Digite um inteiro: 14
Digite um inteiro: 23
    erro! o valor 23 já está na lista
Digite um inteiro: 19
Digite um inteiro: 0

Resultado
[23, 14, 19]
A lista contém 3 elementos
Fim do Programa
```

Neste problema queremos criar uma lista sem valores repetidos. Para implementar a solução é preciso acrescentar uma verificação dentro do laço usando o operador `not in`, conforme mostrado acima. Se o valor não está na lista é acrescentado e se já está a mensagem de erro é exibida.

Exercício Proposto 7.7

Enunciado: Altere a solução ex.resolvido 7.6 de modo que todos os valores repetidos rejeitados sejam incluídos em uma segunda lista. Ao final exiba essa segunda lista juntamente com seu tamanho.

Exercício Resolvido 7.7

Enunciado: Escreva um programa que leia um número inteiro Qtde e carregue uma lista com essa quantidade de números inteiros aleatórios. Exiba a lista na tela.

Em seguida inicie um laço que deve permanecer em execução enquanto um valor inteiro X for maior que zero. Para cada valor de X verifique se ele está ou não na lista gerada. Caso esteja é preciso exibir a quantidade de ocorrências.

```
from random import randint
# primeira parte - geração da lista
Lst = []
Qtde = int(input('Digite a quantidade: '))
for i in range(Qtde):
    a = randint(1, 20)
    Lst.append(a)
print('Lista gerada:')
print(f' {Lst}\n')
# segunda parte - pesquisa na lista
X = 1
while X > 0:
    X = int(input('Digite X: '))
    if X in Lst:
        print(f' há {Lst.count(X)} ocorrência(s) de {X} na lista')
    else:
        print(f' {X} não está na lista')
print('Fim do Programa')
Digite a quantidade: 20
Lista gerada:
[9, 10, 13, 18, 16, 20, 15, 12, 2, 14, 13, 17, 14, 20, 7, 20, 8, 4, 20, 12]

Digite X: 9
há 1 ocorrência(s) de 9 na lista
Digite X: 20
há 4 ocorrência(s) de 20 na lista
Digite X: 3
3 não está na lista
Digite X: 12
há 2 ocorrência(s) de 12 na lista
Digite X: 0
0 não está na lista
Fim do Programa
```

Esta solução tem duas partes. Na primeira é feita a geração e a exibição da lista. Como o enunciado fala em gerar números inteiros aleatórios, foi usada a função `randint()` que pertence ao módulo `Random`, que faz parte da instalação padrão de Python. Os valores foram gerados na faixa de 1 a 20 quando escrevemos `a = randint(1, 20)`, mas você pode mudar esses limites e ver como fica a geração da lista.

Lembrando:

A existência de módulos (foi usado o termo "bibliotecas") foi mencionada no capítulo 1 e sua importação já foi exemplificada no exemplo 2.10

Na segunda parte foi usado o operador `in` para saber se o valor de X está na lista e foi usado o método `.count()` para saber a quantidade de ocorrências.

7.7 TUPLAS – CLASSE `tuple`

Tuplas (classe `tuple`) são sequências imutáveis, usadas com frequência para armazenar sequências de dados heterogêneos, embora nada exista nenhum impedimento para que uma tupla seja homogênea. A tupla é uma das classes compostas mais elementar disponível em Python.

Simplificando um pouco pode-se pensar na tupla como sendo uma lista "só para leitura".

Elas são usadas em situações em que queremos armazenar um número fixo de itens que não serão alterados. Alguns exemplos de situações em que tuplas são usadas:

- representar coordenadas cartesianas (x, y): muito usados em jogos e gráficos;
- cores no formato RGB (vermelho, verde, azul): muito usados em jogos e aplicações web;
- registros em uma tabela de banco de dados (matrícula, nome, idade, endereço, etc): muito usado em sistemas comerciais; e assim por diante.

A principal característica que nos leva a considerar o uso de tuplas nos nossos programas são o fato de que elas consomem quantidades relativamente pequenas de memória em comparação com as listas.

O exemplo 7.16 mostra como uma tupla é criada. Podemos usar dois formatos, com os dados entre parênteses, com no caso das tuplas T e V; ou sem os parênteses como no caso da tupla U.

Este exemplo também mostra que tuplas podem ser heterogêneas (T e U) ou homogêneas (V).

Exemplo 7.16

[Teste este código no Idle](#)

```
>>> T = (12, 'Python', 27.3, 14)
>>> type(T)
<class 'tuple'>
>>> U = 12, 'Python', 27.3, 14
>>> type(U)
<class 'tuple'>
>>> V = (10, 12, 14, 16)
>>> type(V)
<class 'tuple'>
>>> print(T[0])
12
>>> print(T[1])
'Python'
>>> for dado in T:
...     print(dado)
12
Python
27.3
14
>>> T[0] = 26
Traceback (most recent call last):
  File "<pyshell#186>", line 1, in <module>
    T[0] = 26
TypeError: 'tuple' object does not support item assignment
```

(exemplo interativo feito com IDE Idle)

Também é mostrado que se pode acessar cada elemento individualmente usando o índice, bem como usar a tupla em uma iteração com o comando `for`.

E a tentativa de alterar um elemento de T ao fazer `T[0] = 26` resulta em um erro.

O quadro 7.3 mostra como essa classe é leve, quando comparada à classe `list` (ver quadro 7.2 em comparação). Ela contém apenas dois métodos e ambos são relacionados ao acesso aos seus elementos e por consequência quando usada consome menos bytes da memória do computador.

Quadro 7.3 – Métodos da classe `tuple`

Método	Descrição
<code>.count(object)</code>	Retorna o número de ocorrências de um objeto dentro da tupla ex. <code>Qtde = T.count(5)</code>
<code>.index(value, [start, stop])</code>	Retorna o índice da primeira ocorrência do objeto <code>value</code> dentro da tupla. Se <code>start</code> e <code>stop</code> (opcionais) forem fornecidos o método considera apenas seu intervalo. Caso <code>value</code> não esteja na tupla é gerado um erro ex. <code>posição = T.index(5)</code>

Por fim, tuplas e listas são totalmente intercambiáveis, ou seja, é possível realizar conversões entre objetos dessas duas classes. Veja o exemplo a seguir:

Exemplo 7.17

Teste este código no Idle

```
>>> T = (3.6, 120, 'texto')
>>> type(T)
<class 'tuple'> # conferindo que T é uma tupla
>>> T = list(T)   # converte T para lista e armazena no próprio identificador T
>>> type(T)
<class 'list'>  # conferindo que agora T é uma lista
>>> T[0] = 19.25   # alterando um elemento de T. Não ocorre erro porque é lista
>>> print(T)      # conferindo o conteúdo com a alteração
[19.25, 120, 'texto']
>>> T = tuple(T)   # reconvertendo a lista T para tupla
>>> type(T)
<class 'tuple'> # conferindo que T voltou a ser uma tupla
>>> print(T)      # e que manteve o valor alterado
(19.25, 120, 'texto')
(exemplo interativo feito com IDE Idle)
```

Neste exemplo convertemos a tupla `T` para lista, alteramos um de seus valores e retornamos de lista para tupla. Isso é algo que sempre pode ser feito em nossos programas, se for necessário, mas antes de se sentir tentado a fazê-lo pense se é mesmo necessário.

Fazemos esse aviso porque, se muitas conversões entre tuplas e listas precisarem ser feitas, considere usar uma lista de uma vez por todas e não gastar processamento com as conversões em si.

7.8 STRINGS – CLASSE `str`

A classe string é a terceira na categoria das sequências. É uma classe ao mesmo tempo simples e muito importante, pois é muito usada. Afinal, sequências de texto estão presentes em todos os programas que você possa imaginar escrever, basta constatar que já estamos usando strings desde o primeiro exemplo (exemplo 2.1 no cap. 2) escrito neste material. O quadro 7.4 elenca os métodos de uso mais frequente da classe string.

Quadro 7.4 – Métodos da classe `str`

Método	Descrição
<code>.capitalize()</code>	Retorna um string com a primeira letra maiúscula e as demais minúsculas, sem afetar os demais caracteres.
<code>.count(sub, start, end)</code>	Conta quantas vezes um substring <code>sub</code> ocorre dentro do string. Se <code>start</code> e <code>end</code> forem especificados considera apenas os caracteres nesse intervalo.
<code>.find(sub, start, end)</code>	Pesquisa o substring <code>sub</code> dentro da string e retorna um número inteiro indicando a posição se o encontrar, ou -1 caso não encontre. Se <code>start</code> e <code>end</code> forem especificados considera apenas os caracteres nesse intervalo.
<code>.join(iteravel)</code>	Concatena qualquer número de elementos passados no <code>iteravel</code>
<code>.isalnum()</code>	Retorna True caso o string contenha apenas letras e números. Caso contrário, retorna False.
<code>.isalpha()</code>	Retorna True caso o string contenha apenas letras. Caso contrário, retorna False.
<code>.isnumeric()</code>	Retorna True caso o string contenha apenas algarismos. Caso contrário, retorna False. Útil para testar apenas dados numéricos foram digitados.
<code>.lower()</code>	Retorna um string com todas as letras minúsculas e não afeta os demais caracteres.
<code>.lstrip()</code>	Remove espaços em branco do início (lado esquerdo) do string. Também remove caracteres '\n' se presentes. Aqui vale a mesma observação feita para <code>.strip()</code>
<code>.partition(sep)</code>	Pesquisa o string em busca do substring <code>sub</code> . Caso o encontre retorna três strings: a parte antes de <code>sub</code> , o próprio substring <code>sub</code> e a parte depois de <code>sub</code> . Caso não o encontre retorna o próprio string mais dois strings vazios.
<code>.replace(old, new, count)</code>	Procura o substring <code>old</code> e o substitui pelo substring <code>new</code> . Se <code>count</code> for fornecido, apenas as primeiras <code>count</code> ocorrências são substituídas.
<code>.rstrip()</code>	Remove espaços em branco do final (lado direito) do string. Também remove caracteres '\n' se presentes. Aqui vale a mesma observação feita para <code>.strip()</code>
<code>.split(sep)</code>	Quebra um string em partes, usando <code>sep</code> como delimitador entre as partes. Retorna uma lista de strings com as partes. O delimitador <code>sep</code> não é incluído nas partes. Se <code>sep</code> não for fornecido o método usará um espaço em branco como delimitador.
<code>.strip()</code>	Remove espaços em branco do início e do final do string. Também remove caracteres '\n' se presentes. obs: Este método é um pouco mais elaborado que isso. Consulte a documentação para mais detalhes.
<code>.swapcase()</code>	Retorna um string invertendo as letras maiúsculas e minúsculas e não afeta os demais caracteres.
<code>.title()</code>	Retorna um string com a primeira de cada palavra em letra maiúscula e as demais minúsculas. Não afeta os demais caracteres.
<code>.upper()</code>	Retorna um string com todas as letras maiúsculas e não afeta os demais caracteres.

No quadro 7.4 não estão listados todos os métodos disponíveis na classe string. Incluímos os métodos que são usados com mais frequência. Para uma lista completa consulte a documentação em <https://docs.python.org/pt-br/3/library/stdtypes.html#string-methods>

O exemplo 7.18 ilustra o uso do método `.isnumeric()` para validar uma entrada via teclado. É uma forma de evitar erros de digitação de números inteiros.

Exemplo 7.18

Teste este código no PyCharm

```
S = input('Digite um número inteiro: ')
if S.isnumeric():
    N = int(S)
    print(f' o número digitado foi {N} ')
else:
    print(' Erro: digite apenas números')
```

primeira execução

```
Digite um número inteiro: 188
o número digitado foi 188
```

segunda execução

```
Digite um número inteiro: 13abc9
Erro: digite apenas números
```

O exemplo 7.19 ilustra uma situação muito comum, em que vários valores são digitados em uma única linha, devem ser lidos e separados. A ideia é que o usuário possa digitar algo assim:

```
Digite dois inteiros e um real: 53 18 37.9
```

Após a leitura queremos que o valor 53 esteja em um objeto A, o 18 esteja em B e o 37.9 esteja em X. A solução para isso está neste exemplo:

Exemplo 7.19

Teste este código no PyCharm

```
S = input('Digite dois inteiros e um real: ')
L = S.split() # Faz a separação usando espaço em branco como separador
print("lista L: ", L) # Exibe na tela a lista L
A = int(L[0]) # converte o elemento L[0] para inteiro
B = int(L[1]) # converte o elemento L[1] para inteiro
X = float(L[2]) # converte o elemento L[2] para real
print(f'A = {A}, B = {B}, X = {X}')
```

```
Digite dois inteiros e um real: 53 18 37.9
lista L: ['53', '18', '37.9']
A = 53, B = 18, X = 37.9
```

Adiante neste curso teremos vários usos para objetos da classe `str` e seus métodos. À medida que precisarmos de algo que ainda não foi abordado faremos uma explicação específica e demonstração de uso.

Exercício Resolvido 7.8

[Teste este código no Python](#)

Enunciado: Escreva um programa que leia um string contendo três números inteiros separados por espaços em branco. Separe-os em objetos int, calcule sua soma e exiba na tela.

Entrada: 26 128 44 (string com três inteiros separados por espaço em branco)

Saída: n1 = 26

n2 = 128

n3 = 44

Soma = 198

```
valores = input('Digite 3 inteiros separados por espaço: ')
valores = valores.split() # usa o método .split() para separar os valores
for i in range(len(valores)): # itera sobre a lista valores
    valores[i] = int(valores[i]) # e converte cada objeto para int
print(f'n1 = {valores[0]}')
print(f'n2 = {valores[1]}')
print(f'n3 = {valores[2]}')
print(f'Soma = {sum(valores)}') # calcula e exibe a soma
Digite 3 inteiros separados por espaço: 26 128 44
n1 = 26
n2 = 128
n3 = 44
Soma = 198
```

Neste exemplo usamos o método `.split()` para separar o string, usando o caractere espaço em branco como critério de separação. Como resultado obtemos uma lista onde cada elemento é também um string. Em seguida cada um é convertido para número inteiro e exibido na tela. No final a soma dos elementos da lista é calculada e também exibida.

7.9 CARACTERÍSTICAS COMUNS ÀS CLASSES DE SEQUÊNCIAS – LISTAS, TUPLAS E STRINGS

Em síntese, essas são as características mais relevantes dos objetos de sequências de Python

- **Ordenados:** contêm elementos organizados sequencialmente de acordo com sua ordem de inserção específica.
- **Indexáveis** através de um índice baseado em zero: Permitem acessar seus elementos por índices inteiros que começam em zero.
- **Aninháveis:** Lista e tuplas podem conter outras sequências. Desse modo que é possível criar lista de listas, lista de tuplas, tupla de tuplas e tupla de listas.
- **Iterável:** Eles suportam iteração, então você pode percorrê-los usando um loop ou compreensão enquanto executa operações com cada um de seus elementos.
- **Fatiáveis:** Eles suportam operações de fatiamento, o que significa que você pode extrair uma série de elementos de uma sequência qualquer.
- **Combináveis:** Eles suportam operações de concatenação, portanto você pode combinar duas ou mais tuplas usando os operadores de concatenação, o que cria uma nova sequência.

Exercício Proposto 7.8

Enunciado: Escreva um programa que leia um string contendo uma data no formato `aaaammdd`, onde `aaaa` é o ano com 4 dígitos; `mm` é o mês com 2 dígitos; `dd` é o dia com 2 dígitos. O programa deve validar a entrada verificando dois itens: a) se foram fornecidos 8 caracteres; b) se todos os caracteres são dígitos numéricos. Se a entrada for válida o programa deve produzir a saída exemplificada abaixo; se for inválida deve exibir uma mensagem de erro (que você pode elaborar como desejar).

Saída: A data fornecida é: `dd/mm/aaaa`

Dica: Use fatiamento

Exercício Proposto 7.9

Enunciado: Altere o programa do Exercício Proposto 7.8 acrescentando uma validação adicional para garantir que a data fornecida seja válida. Por exemplo: a entrada `20242255` é válida segundo os critérios estabelecidos no enunciado 7.8. Porém, `55/22/2024` não é uma data válida.

Neste exercício você deve garantir que a data seja válida (incluindo anos bissextos – para identificar se um ano é bissexto veja o Exercício Proposto 4.8).

Exercício Proposto 7.10

Enunciado: Escreva um programa que leia um número inteiro `Qtde` e carregue uma lista com essa quantidade de números inteiros aleatórios entre 1 e 100. Exiba a lista na tela.

Em seguida inicie um laço que deve permanecer em execução enquanto um valor inteiro `X` for maior que zero. Para cada valor de `X` verifique se ele está na lista gerada e caso esteja elimine-o. Se houver mais de uma ocorrência de `X` na lista, elimine todas. Após as eliminações exiba a lista novamente.

Exercício Proposto 7.11

Enunciado: Escreva um programa que leia um número inteiro Qtde e carregue uma lista com essa quantidade de números inteiros aleatórios quaisquer. Exiba a lista na tela.

Em seguida verifique se existem e elimine valores que estiverem repetidos, deixando apenas uma ocorrência de cada. A ordem relativa dos elementos na lista não deve ser alterada, com exceção às consequências da eliminação dos repetidos. Exiba a nova lista sem repetidos e o seu tamanho.

Exercício Proposto 7.12

Enunciado: Escreva um programa que permaneça em laço de modo que em cada repetição seja lido e armazenado em uma lista o nome de uma pessoa. O laço termina quando o usuário entrar com um string vazio.

Exiba na tela a lista de nomes em ordem alfabética e precedida de um número de ordem começando em 1.

Exemplo:

- 1 Bernardo Almeida
- 2 Carlos Eduardo Soares
- 3 Julia Monteiro da Silva
- 4 Margarete Guimarães
- 5 Robson de Souza Andrade

Exercício Proposto 7.13

Enunciado: Escreva um programa que permaneça em laço lendo três dados de um produto: o código (int), o preço de compra (float) e o preço de venda(float). Com esses dados forme uma tupla e armazene-a em uma lista. Os três dados devem ser lidos em uma única linha separados por espaço em branco.

O laço termina quando forem digitados três zeros: 0 0 0

Em seguida, para todas as tuplas presentes na lista, exiba o código do produto e a margem bruta de lucro do produto em porcentagem e com uma casa decimal.

A margem bruta de lucro é calculada com a expressão:

$$\text{MargemBruta} = \left(\frac{\text{Preço Venda}}{\text{Preço de Compra}} - 1 \right) \cdot 100\%$$

Exercício Proposto 7.14

Enunciado: Escreva um programa que leia um número inteiro nA e gere uma lista A com nA valores inteiros aleatórios, não repetidos e situados na faixa [1, 100]. Mostre-a na tela em ordem crescente.

Em seguida leia outro inteiro nB e gere a lista B usando as mesmas regras aplicadas à lista A. Mostre-a na tela também em ordem crescente.

Crie e exiba uma lista contendo a união das listas A e B, sem conter valores repetidos. Mostre a lista resultante e a quantidade de elementos dela.

Exemplo: nA = 7 lista A = [8, 12, 29, 35, 44, 64, 81]

nB = 5 lista B = [10, 25, 35, 38, 64]

Saída: União de A e B

[8, 10, 12, 25, 29, 35, 38, 44, 64, 81] contém 10 elementos

